(12) UK Patent Application (19) GB (11) 2 366 499 (13) A

(43) Date of A Publication 06.03.2002

(21) Application No 0021081.5

(22) Date of Filing 25.08.2000

(51) INT CL[7]
G06F 17/30

(52) UK CL (Edition T)
H4T TDXX

(56) Documents Cited
EP 0944009 A2 EP 0859330 A1
WO 98/44434 A1

(58) Field of Search
UK CL (Edition S) H4F FDX, H4T TBEC TBEX TBLA TBLX TDXX
INT CL[7] G06F 17/30
ONLINE: WPI; EPODOC; JAPIO; INSPEC; XPESP; COMPUTER

(71) Applicant(s)
Copyn Limited
(Incorporated in the United Kingdom)
10th Floor, One America Square, Crosswall,
LONDON, EC3N 2PR, United Kingdom

(72) Inventor(s)
Geraint Edwards
Christopher Needham

(74) Agent and/or Address for Service
Reddie & Grose
16 Theobalds Road, LONDON, WC1X 8PL,
United Kingdom

(54) Abstract Title
A method of storing a portion of a web-page

(57) Portions of mark-up language pages may be stored in an on-line repository. The user selects a portion of a page for storage using a pointer device and an extension to a browser context menu. If the mark-up code for the selected portion corresponds to a predefined meaningful element, the DOM node to which it refers is identified and the node tree traversed to look for meaningful collections of elements, the raw HTML is then extracted and sent to a new window where it can be selected and stored in a remote database. The database is configured to enable a scrapbook like presentation of displayed elements with elements displayed as cards. Cards may be stored in a number of leaves and card parameters, and leaf configurations may be customised by a user. Access rights can be granted to allow elements in a given repository to be viewed by others.

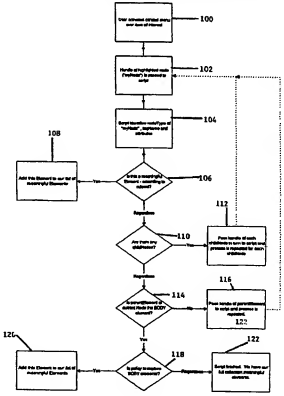

Figure 7

GB 2 366 499

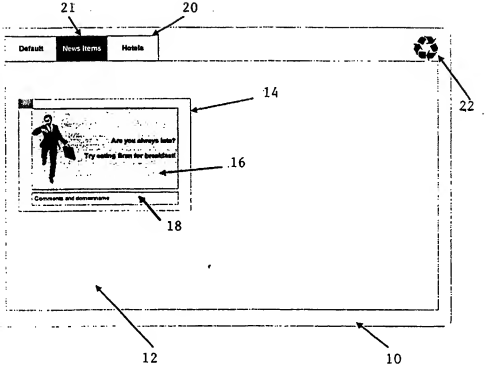
21
20
Default
News Items
Hotels
14
22
Are you always late?
Try eating Bran for breakfast
16
Comments and commentaries
18
12
10

FIGURE 1

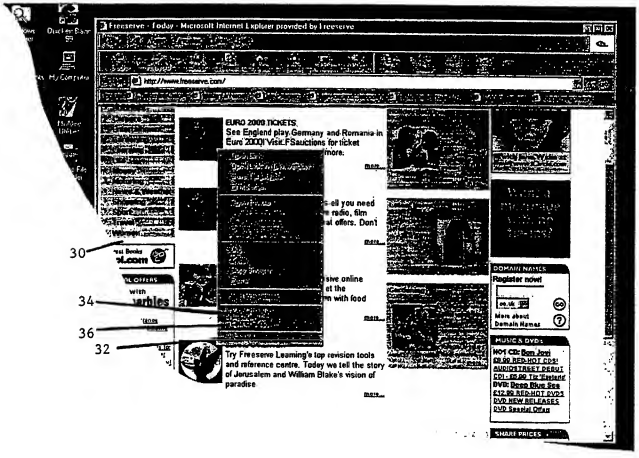
Freeserve - Today - Microsoft Internet Explorer provided by Freeserve
http://www.freeserve.com/
EURO 2000 TICKETS.
See England play Germany and Romania in Euro 2000! Visit FSauctions for ticket
more.
more.
more.
more.
DOMAIN NAMES
Register now!
Main menu
Domain Names
MUSIC & DVDs
HOT CD: Bon Jovi
CD £9.99 RED-HOT CDs!
AUDIO TARGET PUBLI
CD1 - £9.99 Tin Machine
DVD: Bryan Ohya Sea
£12.99 RED-HOT DVDs
DVD NEW RELEASES
DVD Special Offers
SMART PRICES
Try Freeserve Learning's top revision tools and reference centre. Today we tell the story of Jerusalem and William Blake's vision of paradise.
more...
30
34
36
32

FIGURE 2

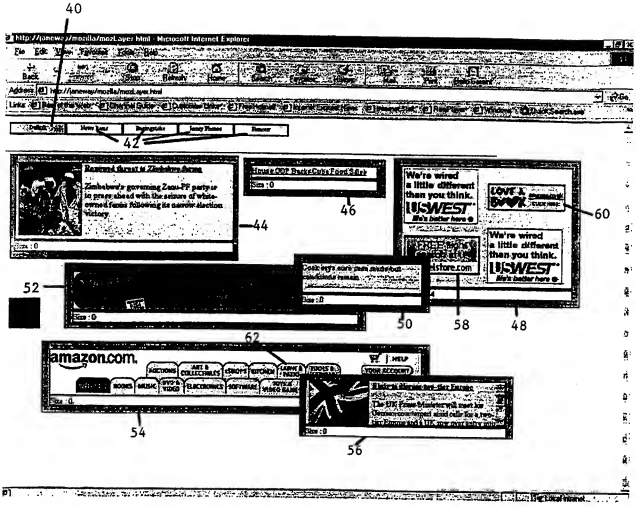

FIGURE 3

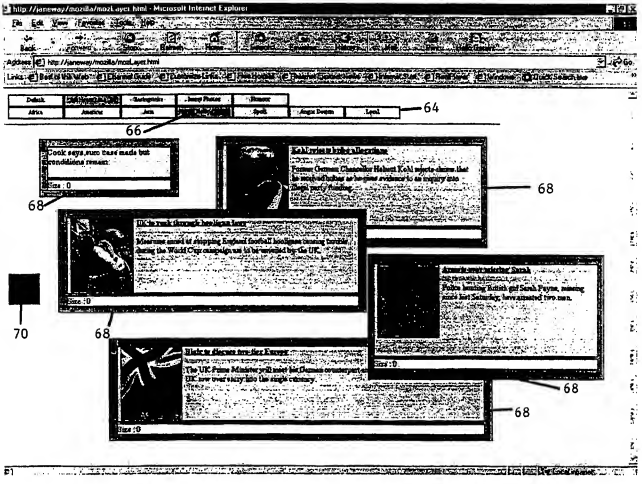

FIGURE 4

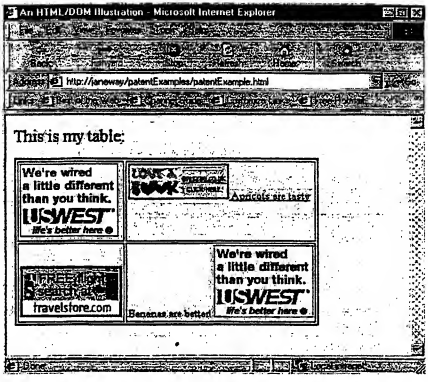
An HTML/DOM Illustration - Microsoft Internet Explorer
Address
http://areaway/patentExamples/patentExample.html
This is my table:
We're wired a little different than you think.
USWEST
We're better here
Avocados are tasty
travelsfore.com
Bananas are better
We're wired a little different than you think.
USWEST
We're better here

FIGURE 5

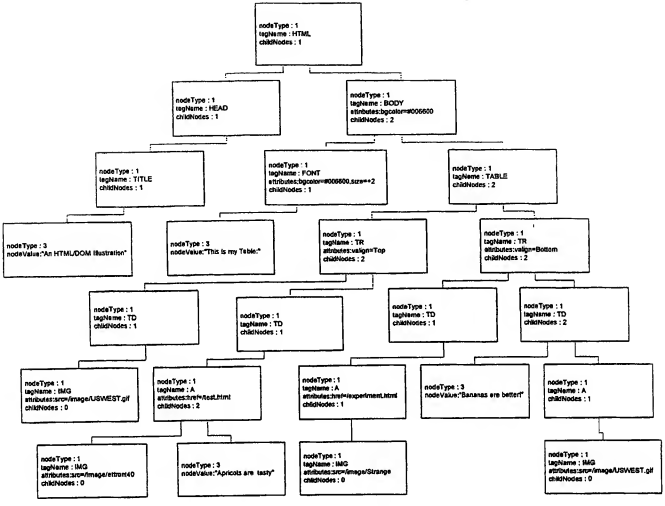
nodeType : 1
tagName : HTML
childNodes : 1
nodeType : 1
tagName : HEAD
childNodes : 1
nodeType : 1
tagName : BODY
attributes:bgcolor=#006600
childNodes : 2
nodeType : 1
tagName : TITLE
childNodes : 1
nodeType : 1
tagName : FONT
attributes:bgcolor=#006600,size=+2
childNodes : 1
nodeType : 1
tagName : TABLE
childNodes : 2
nodeType : 3
nodeValue:"An HTML/DOM illustration"
nodeType : 3
nodeValue:"This is my Table:"
nodeType : 1
tagName : TR
attributes:valign=Top
childNodes : 2
nodeType : 1
tagName : TR
attributes:valign=Bottom
childNodes : 2
nodeType : 1
tagName : TD
childNodes : 1
nodeType : 1
tagName : TD
childNodes : 1
nodeType : 1
tagName : TD
childNodes : 1
nodeType : 1
tagName : TD
childNodes : 2
nodeType : 1
tagName : IMG
attributes:src=/image/USWEST.gif
childNodes : 0
nodeType : 1
tagName : A
attributes:href=/test.html
childNodes : 2
nodeType : 1
tagName : A
attributes:href=/experiment.html
childNodes : 1
nodeType : 3
nodeValue:"Bananas are better!"
nodeType : 1
tagName : A
childNodes : 1
nodeType : 1
tagName : IMG
attributes:src=/image/ettrtnt40
childNodes : 0
nodeType : 3
nodeValue:"Apricots are tasty"
nodeType : 1
tagName : IMG
attributes:src=/image/Strange
childNodes : 0
nodeType : 1
tagName : IMG
attributes:src=/image/USWEST.gif
childNodes : 0

FIGURE 6

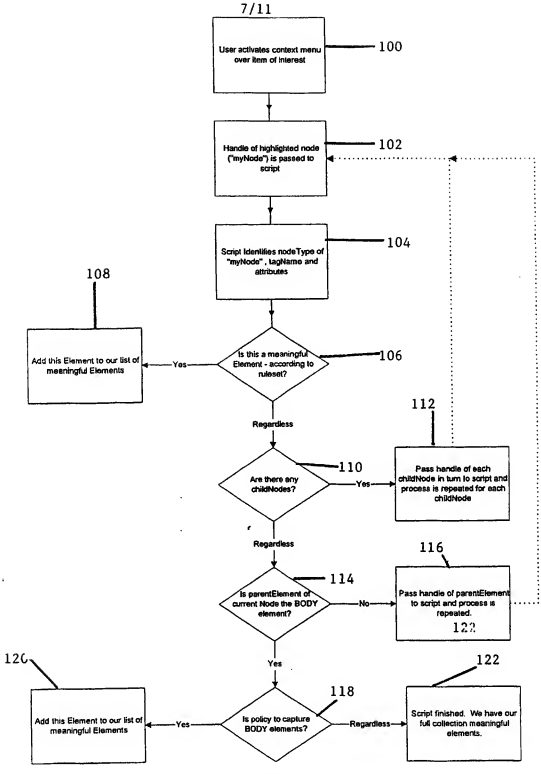
User activates context menu over item of interest
100
Handle of highlighted node ("myNode") is passed to script
102
Script identifies nodeType of "myNode", tagName and attributes
104
108
Add this Element to our list of meaningful Elements
Yes
Is this a meaningful Element - according to ruleset?
106
Regardless
112
Pass handle of each childNode in turn to script and process is repeated for each childNode
Are there any childNodes?
110
Yes
Regardless
116
Is parentElement of current Node the BODY element?
114
No
Pass handle of parentElement to script and process is repeated.
122
Yes
120
Add this Element to our list of meaningful Elements
Yes
Is policy to capture BODY elements?
118
Regardless
122
Script finished. We have our full collection meaningful elements.

Figure 7

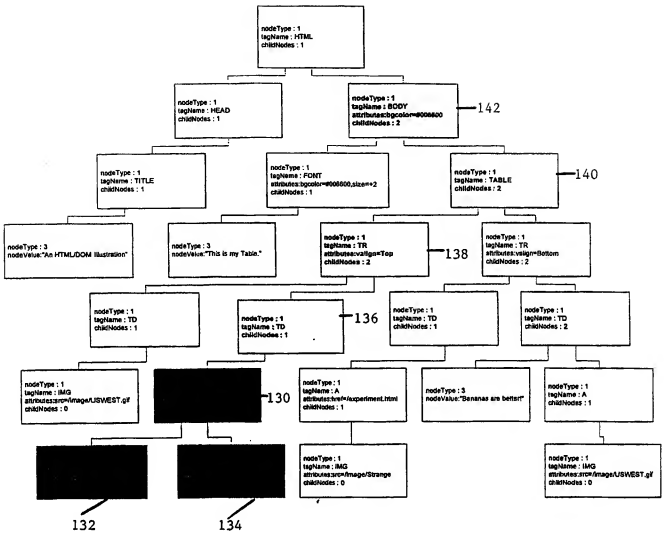
nodeType : 1
tagName : HTML
childNodes : 1
nodeType : 1
tagName : HEAD
childNodes : 1
nodeType : 1
tagName : BODY
attributes:bgcolor=#006600
childNodes : 2
142
nodeType : 1
tagName : TITLE
childNodes : 1
nodeType : 1
tagName : FONT
attributes:bgcolor=#006600,size=+2
childNodes : 1
nodeType : 1
tagName : TABLE
childNodes : 2
140
nodeType : 3
nodeValue:"An HTML/DOM Illustration"
nodeType : 3
nodeValue:"This is my Table."
nodeType : 1
tagName : TR
attributes:valign=Top
childNodes : 2
138
nodeType : 1
tagName : TR
attributes:valign=Bottom
childNodes : 2
nodeType : 1
tagName : TD
childNodes : 1
nodeType : 1
tagName : TD
childNodes : 1
136
nodeType : 1
tagName : TD
childNodes : 1
nodeType : 1
tagName : TD
childNodes : 2
nodeType : 1
tagName : IMG
attributes:src=/image/USWEST.gif
childNodes : 0
130
nodeType : 1
tagName : A
attributes:href=/experiment.html
childNodes : 1
nodeType : 3
nodeValue:"Bananas are better!"
nodeType : 1
tagName : A
childNodes : 1
nodeType : 1
tagName : IMG
attributes:src=/image/Strange
childNodes : 0
nodeType : 1
tagName : IMG
attributes:src=/image/USWEST.gif
childNodes : 0
132
134

FIGURE 8

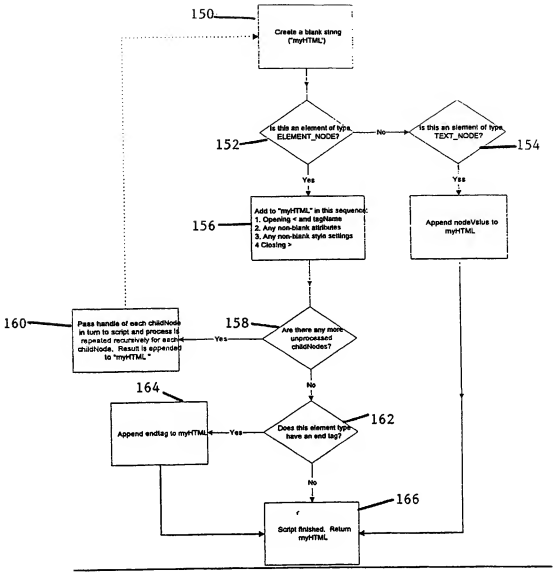
150
Create a blank string ("myHTML")
Is this an element of type ELEMENT_NODE?
No
Is this an element of type TEXT_NODE?
154
152
Yes
Yes
Add to "myHTML" in this sequence:
1. Opening < and tagName
2. Any non-blank attributes
3. Any non-blank style settings
4 Closing >
156
Append nodeValue to myHTML
160
Pass handle of each childNode in turn to script and process is repeated recursively for each childNode. Result is appended to "myHTML"
158
Are there any more unprocessed childNodes?
Yes
No
164
Append endtag to myHTML
Yes
Does this element type have an end tag?
162
No
166
Script finished. Return myHTML

FIGURE 9

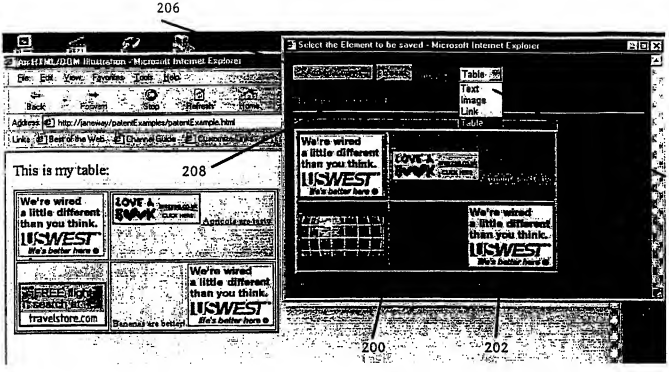
206
Select the Element to be saved - Microsoft Internet Explorer
Table
Text
Image
Link
Table
This is my table:
208
We're wired a little different than you think.
USWEST
We're better here
LOVE A
travelstore.com
Business are better!
200
202
204

FIGURE 10

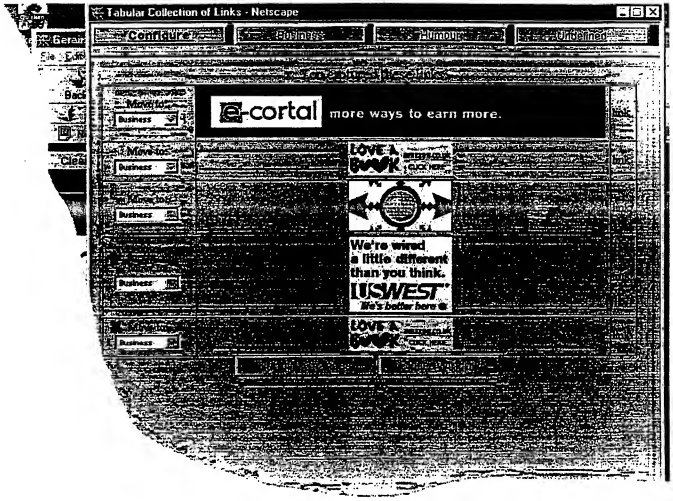
Tabular Collection of Links - Netscape
Contours
Business
Business
Business
Business
Business
Business
e-cortal
more ways to earn more.
LOVE A BOOK
We're wired a little different than you think.
USWEST
We're better here

FIGURE 11

Capture, Storage and Retrieval of Markup Elements

This invention relates to the retrieval of content from the Internet , and particularly to the storage and retrieval of that content.

World wide Web browsers, such as Netscape Navigator, hereafter referred to as NN, and Internet Explorer, hereafter referred to as IE, provide functionality to aid the web browsing experience. The creators of web browsers recognise that users have particular pages that they wish to revisit, and so incorporate functionality to allow the user to add a page to their "favorite" (IE) or "bookmark" (NN) list. This list is stored on the user's computer (or network file system) in a tree-like hierarchy, enabling the user to create a simple classification of information. Each favorite or bookmark is represented by a text description (and in some circumstances a small icon). Users can customise the description of each favorite/bookmark, to a limited extent, with the default being the title of the page. In the latest version of IE, version 5, the user can change the icon associated with a favorite, but this is somewhat cumbersome; the default option is to use an icon provided by the web publisher.

Although the bookmark and favorite options are useful, they suffer from a number of disadvantages. There is no mechanism for informing users when pages in the favorites/bookmarks list have become stale or their content has changed significantly. However, there is a function in IE which allows the user to make a copy of a web page and store it off-line, in which case IE can inform the user if the content of the online version has changed from that of the stored copy.

The context menu, obtained by clicking the right mouse button over a specific item on a page in the MS Windows

operating system provided by Microsoft Corporation, enables the user to save the link associated with that individual item. In NN, the user can bookmark the link associated with an image, and also save the image itself; however the link and the image are stored as two separate entities. The context menu is launched by different methods in different operating systems.

Furthermore, there are no mechanisms to enable the easy access of bookmarks/favorites from different computers, or to share them with other people, or for a number of different users to work collaboratively on them. However, to a limited extent, and in a cumbersome way, these things can be achieved in part by using the import and export functions for bookmarks/favorites.

Recognising the limitations of the favorite/bookmark functionality of web browsers, a number of companies have created alternative services and products that attempt to improve on certain aspects of the browser functionality:

Backflip, Blink and HotLinks, whose products are available at: www.backflip.com, www.blink.com, www.hotlinks.com each provide an online implementation of the basic browser bookmark/favorite functionality, together with organisation and search capabilities.
The main benefits are that users can access their bookmarks from any computer and, if they choose, share them with other people. The main way of activating the service is for the user to register online and download a simple DHTML scriptlet, which adds the functionality to the user's browser, and adds "Backflip", "Blink" or "HotLinks" buttons to the personal tool bar (IE) or link bar (NN). The other way is for web publishers to opt-in to the service and display "Backflip", "Blink" or "HotLinks" button to the personal tool bar, IE, or the link bar, NN. The scriptlet does nothing more than

determine the URL of the page being read and send it to a server. The other way is for web publishers to opt-in to the services and display "Backflip", "Blink" or "Hotlinks" icons on their web pages, which a user can click to save a given page to their online collection. Hotlinks can also tell users if pages have expired or are no longer available.

Although these services address some of the disadvantages of browsers discussed above, the user is limited to bookmarking whole pages or frames, rather than links or images within a page.

Yahoo! Companion provided by Yahoo, Inc. and available at docs.companion.yahoo.com is a package of services, a feature of which is called Y!Bookmarks which, like Backflip, Blink or HotLinks, is an online implementation of the basic browser bookmark/favorite functionality. The service is activated by the user to registering online and downloading a plug-in, which adds the functionality to the user's browser. The new functionality manifests itself as a whole new tool bar which includes a Y!Bookmarks button, amongst others. While the implementation is more sophisticated than the other on-line services mentioned above, the benefits and limitations of Y!Bookmarks are similar to those of Backflip, Blink or HotLinks.

clicVu, which can be found at www.clicVu.com is a service which enables users to save banner adverts in an online collection of banner adverts. The main benefit of this service arises from the fact that, on any given web page, the banner advert(s) are regularly refreshed, so bookmarking the whole page does not save the particular advert of interest to the user. Another benefit of the service is that the bookmarked items are represented visually in the user's online collection using the original banner advert images. This service requires the

advertiser/publisher to opt-in and display a "clicVu" icon on their banner adverts, on which the user clicks. Unlike Backflip, Blink, HotLinks and Y!Companion, the user does not have to register with clicVu (though only limited features of the service are available to users who do not register). The service has the disadvantage that it is limited to banner adverts, and only those where the relevant advertiser/publisher has opted in to the service. It allows users to save one specific type of element and it does not save generic HTML (Hyper Text Mark Up Language) elements, requiring the publisher/advertiser to opt in. What is saved in the user's collection is not under the control of the user.

Visual Bookmarks, available at www.visualbookmark.com is one of a small number of bookmark services that associate images with bookmarks. In each case the image is a full or partial windows screen dump of the browser window - in other words it is a static bitmap representation of the page. Any web links associated with these static images will be set to the URL of the page.

The above examples are all either browser or on-line services. In addition, the following systems, while not bookmarking technologies use related concepts.

Napster, available at www.napster.com is a service that allows users to make their MP3 files available to other users online and to search for music files in which they may be interested. It is a combination of a searchable directory and a tool that users can download to make MP3 files on their hard disks available on the web (even if they are not running a web server on their machine). Although not strictly a bookmarking service, by adding their entries to a public directory it could be considered to be a form of public 'bookmarking' for MP3 files.

The Windows type operating environment provides a wide number of WYSIWYG (What You See is What You Get) operating environments for computer users, including Microsoft Windows, MacOS, KDE (under X-windows).

These allow the free positioning of 'windows' on the user's screen with an element of memory associated with them. Applications can 'remember' were sub-windows are when they are closed and reopened. These systems, within constraints, allow users to cut and paste items from one application to another. This is not possible with the dynamic content of web-pages beyond a very limited pasting of URLs as HTML links. In some applications, for example, MS Word available from Microsoft Corp., it is possible to cut and paste individual items from a web page, for example an image, but this creates a local copy and cannot be free-positioned or manipulated.

Storing traditional bookmark files on a network file system, (with appropriate user permissions set), allows a limited form of collaboration and some machine independence for users. However, this will often not work well on networks employing more than one operating system such as Unix and Windows because of file permission difficulties.

It will be seen from the above discussion that all existing systems for accessing frequently visited web sites suffer from some or all of a number of disadvantages.

All the prior art systems have the disadvantage that only a very limited number of items types can be "bookmarked". The only items that can be bookmarked by any combination of the products available at present are:

The location of a whole page or frame;

A text based link; or

Banner advertisements. The latter is only possible with clicVu when the advertiser has opted in.

Moreover, representation is always tabular or hierarchical. Furthermore, representations of bookmark/favorites is generally text based, the only exceptions being clicVu which is limited to one specific type of image and then only when the publisher/advertiser opts in, and Visual Bookmarks which uses a static bitmap of the user's screen to represent the link. The prior art has the further disadvantage that the representation is static, in that there is no way to resize or reposition bookmarked elements.

The online services referred to have the further disadvantage that they can only capture full page URLs, not even text based links are possible.

The browser based services have the further disadvantage that they do not store bookmarks online for easy access from many locations;

There are no collaboration capabilities, apart from sharing a set of bookmarks over a network, and

Bookmarked elements can become "stale" with no warning or way of checking other than opening each bookmark in turn.

The present invention, in its various aspects aims to overcome the above mentioned disadvantages and to provide improved storage of web page elements for retrieval by users.

According to a first aspect of the invention, there is provided a method of storing a portion of a mark-up language page, comprising the steps of: identifying, from

a visual representation of the page, a portion of the visual representation of the mark-up language page to be stored; identifying a list of candidate mark-up elements from a predefined set of elements for storage; selecting elements from the list; and storing the selected elements.

The invention also provides apparatus for storing a portion of a mark-up language page, comprising: means for identifying, from a visual representation of the page, a portion of the visual representation of the mark-up language page to be stored; means for identifying a list of candidate mark-up elements from a predefined set of elements for storage; means for selecting elements from the list; and means for storing the selected elements.

Embodiments of the invention have the advantage that any meaningful portion of a website can be selected and bookmarked. For the avoidance of doubt, the term "bookmarked" is used to convey the intention of making a note of the location of an item for subsequent retrieval and is not limited by the prior art. Preferably, the selection of the identified portion comprises selecting an Internet browser context menu and selecting a command from the menu.

Preferably, identifying a list of candidate mark-up elements comprises identifying the node of the document object model which represents the selected portion and extracting the markup code for the identified node and storing that markup code. The markup code may be in HTML or any other suitable markup code such as XML.

Identifying the node includes traversing the node tree of the DOM and identifying ancestor and descendent nodes representing markup elements in the set of predefined set of markup elements.

Node tree traversal may also include establishing a list of markup elements from the predefined set. Node tree traversal may also comprise determining from a predefined rule set whether a given node represents the end of a node tree traversal in a given direction.

The preferred embodiments of the invention allow the capture of any generic meaningful element or meaningful collections of elements at the users selection. This does not require the publisher of the web page in question to subscribe to any service or to opt-in and is wholly independent of the publisher.

This preferred embodiment has the advantage that the elements can be viewed in a free-form non-hierarchical manner which presents a far more user-friendly view to the user. The user can see the visual representation of the actual elements stored and not simply a text heading or the like.

Preferably, the repository comprises a plurality of cards, each card comprising a visual representation on screen of a stored identified portion.

Preferably, the cards are arranged into leaves, each leaf comprising at least one card.

Preferably, the cards are moveable around the leaves.

Preferably, each card may form a part of one or more leaves.

Preferably, a plurality of leaves may be arranged into views, each view comprising a set of identified web page portions and their attributes.

Preferably, a given leaf may form a part of a plurality of views.

The preferred embodiments of the aspect of this invention permit the user a wide degree of flexibility including the ability to cross-reference, define their own categorisation options and their own display options.

Preferably, access parameters may be defined whereby access to a user's stored web page portions may be limited to the user, available to any third party or partially restricted according to the access parameters.

This preferred embodiment has the advantage that the user has complete flexibility over who can see his stored portions.

Preferably, the repository comprises database for storing mark up elements chosen from a set of defined acceptable mark up elements and representing portions of a web page, the database comprising a plurality of tables including an element data table for storing data about the mark-up elements; a card data table storing information about the display, formatting and positioning of the element data stored in the element data table; a leaf data table for storing data regarding cards which can be displayed in a common leaf; and a view data table for storing data about collections of leaves.

The structure embodying the invention allows the complete flexibility in the display, categorisation and cross referencing of stored web page portions referred to above.

Embodiments of the invention will now be described, by way of example, and with reference to the accompanying drawings, in which:

Figure 1 is a pictorial representation of the terminology used to describe embodiments of the invention, for ease of understanding;

Figure 2 is a portion of a sample web page having a context menu overlaid;

Figure 3 is a view of a leaf having a number of cards;

Figure 4 is a view of a sub-leaf;

Figure 5 is a view of a sample web page;

Figure 6 is a view f the Document Object Model (DOM) of the web page of Figure 5;

Figure 7 is a flow diagram illustrating a process for identifying meaningful elements from the DOM;

Figure 8 shows how the DOM tree of Figure 7 may be transversed when identifying meaningful elements;

Figure 9 is a flow diagram illustrating a process for extracting HTML code for identified meaningful elements;

Figure 10 is a screen print showing how an element may be selected for saving; and

Figure 11 is a view of a repository/user interface according to a second embodiment of the invention.

In order to understand the invention it is useful first to review the technical framework underpinning it.

When a user of the Internet browses a web page using one of the available 'web browsers' such as Netscape Communicator (NN) or MS Internet Explorer (IE), the page they see on their screen is actually a rendition of a stream of data presented to the browser in HTML format. HTML (Hyper Text Markup Language), the language of the world wide web, consists of combinations of tags, attributes, such as size, and data/text, which are interpreted by the browser to create a potentially interactive display of information, that appears fairly

similar across all operating systems (such as MS Windows, MacOS or Unix) and different browsers. The whole of a web page need not come from the same server. HTML tags allow the publisher of a web page to merge elements from different sources. In one of its most complicated manifestations, a web portal (such as my.yahoo.com), may bring in elements from many third parties - news stories from one company, stock prices from another and weather forecasts from yet another. They may also be selling part of their page to an advertising server that constantly changes the banner advert the user sees. Often, all of this information is retrieved directly by the user's machine without passing through the publisher's server. In other words, the web publisher can merely point the user to the locations of the various elements of the page and allow the user's machine to obtain the information directly.

The source of a page being viewed by the user is usually dynamic in its content - for example, the front page of a newspaper's web site will be constantly changing. Occasionally pages change so frequently that some items seen on a page (such as a banner advertisement) may never be seen again by the user if they do not respond to them before the page is refreshed or changed; and even a summary of news articles on a web portal will be changing such that an interesting news story may be difficult to retrieve if it is not read at once.

HTML 4.01 is an SGML (Standard Generalised Mark Up Language) application conforming to International Standard ISO 8879 - Standard Generalized Markup Language. The full specification is available from the World Wide Web Consortium (W3C) and the detailed HTML 4.01 Specification Recommendation at is to be found at http://www.w3.org/TR/html401.

Within this specification of HTML 4.01 is the Document Type Definition ("DTD") that defines the markup language within the SGML framework. This document will be used to determine some of the rules followed by the embodiments to be described.

ECMAScript (International Standard ISO/IEC 16262) is a standardised scripting language based in large part on Javascript (Netscape) and Jscript (Microsoft). A detailed description of the language is published by ECMA in the ECMS-262 Ed. 3 standard at http://www.ecma.ch/ecma1/stand/ecma-262.htm.

CSS2 (or Cascading Style Sheets, level2) describes a style sheet language which allows authors and users to attach 'style' (fonts, spacing, placement, size etc.) to structured documents, including HTML documents and XML (Extensible Mark Up Language) applications. The latest W3C (World Wide Web Consortium) recommendation for CSS2, may be found at http://www.w3.org/TR/REC-CSS2.

The Document Object Model (DOM) Level 2 Specification defines a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The DOM Level 2 is made of a set of core interfaces to create and manipulate the structure and content of a document, and a set of optional modules containing specialised interfaces dedicated to XML, HTML , traversing the document etc.

The DOM Level 2 Specification is believed to be close to a recommendation stage and the latest version is published at http://www.w3.org/TR/DOM-Level-2.

The relationship between the DOM and the underlying HTML will be described later in the document.

The Extensible Markup Language (XML) is a subset of SGML that is completely described in the W3C recommendation of February 1998. The recommendation can be found at http://www.w3.org/TR/1998/REC-xml-19980210. XML is supplemented by a raft of other specifications about how the markup language is interpreted visually and how it can be manipulated by scripting languages for example. Note that each XML document will be accompanied by a DTD (since HTML 4.01 is as a specific case of XML it has its own DTD as was mentioned earlier).

To implement embodiments of this invention familiarity is required also with SQL/relational databases, Web server, and CGI/Perl or another interactive web server scripting or programming interface.

The following description relates to an embodiment developed to run on Microsoft's Internet Explorer browser IE (version 5) and Netscape's browser NN (release 6). It uses the ability of browsers to be customised by an application developer. Implementation in other browsers (such as Opera) requires a different user interface but the core mechanics of the underlying invention is the same. Such browsers need to be compliant with the standards described earlier.

This description relates to the latest major released version of the Microsoft's Internet Explorer web browser Version 5 (IES) and the preview release of Netscape Navigator 6 NN6. These browsers have many subtle differences in their implementation of the standards described often using slightly different names for variables or functions. The embodiments to be described can be implemented in either browser; minor differences in

functionality exist that allow differing enhancements to be applied in each environment.

Microsoft's Internet Explorer browser (version 4 onwards) allows developers to add custom items to the context menu; a pop-up menu that appears on the user's screen when he clicks the right mouse button. The context mouse button is accessed slightly differently in the MacOs System. A detailed explanation of the customisation of the context menu is now available from the Microsoft Corporation at their web site http://msdn.microsoft.com/workshop/browser/ext/tutorials/context.asp

Netscape Navigator 6 provides a lot more flexibility to the developer to customise the browser but the process is a little more involved. Almost any part of the NN6 interface can be customised by adding or modifying XUL (XML based user interface language) overlay file and providing or modifying an associated script to the applications "chrome". A chrome in mozilla, the open source browser development project of Netscape Corp, is a complete front end, including all aspects of graphics, layout and functionality. The concepts are explained at http://mozilla.org/xpfe/xptoolkit/overlays.html and http://mozilla.org/xpfe/xptoolkit/popups.html.

An embodiment of the invention will now be described.

Referring now to Figure 1, some terminology will first be described.

An Element of a web page is defined as an HTML tag, or a meaningful collection of HTML tags, which can be saved. An element is likely to include the URL of an item of interest to a user, rather than a copy of the item itself. Examples of Elements include:

A banner advert; a link; an image, with or without an associated link; an MPEG video; an MP3 sound file; and a table of images, which is an example of a meaningful collection of elements being classed as an Element.

A Repository is defined as an online database in which bookmarked Elements are stored. Each user can have one or more repositories.

A Card in the repository is defined as the visual representation on screen of a bookmarked element. It is customisable, but typically it looks like the original element from the original web page, surrounded by a rectangular border.

A Leaf in the repository is defined as the visual representation on screen of a set of cards. It looks like a page from a scrapbook with an index tab attached.

A View is defined as one way of categorising a set of some, or all, of the bookmarked elements in the Copyn repository together with their attributes such as position on screen, size, background colour etc. and the attributes of the leaves on which they are displayed. For any given set of Elements, that is a Repository, there can be many different Views. Views are made up of a collection of Leaves.

In the following description and claims, no distinction is made between the visual representation on screen of cards and leaves and the underlying mark-up data or its DOM representation. This is because the visual representation is the direct result of a web browser, or other such computer program, interpreting the mark-up data representation, or its DOM equivalent, of the card or leaf

and generating the resultant visual image and behaviour on screen. Hence when it is stated that a card is movable on screen, it means that the underlying mark-up language or DOM equivalent is modified such that the web-browser, or other program, displays the card in another position. In addition it means that a user interface is provided, via the browser or the like, such that the underlying mark-up, or DOM equivalent can be manipulated.

Thus, in Figure 1, a browser window is shown generally at 10. Within the browser window 10 is shown a leaf 12 which contains cards. One such card is shown at 14 although typically a leaf would contain several cards. The card contains an element 16 which comprises a meaningful HTML element as described above. The card also includes a space 18 for inclusion of a user defined comment and domain name and other text. The leaf is one of a number of leaves in the repository and each leaf can be accessed by clicking on a leaf index tab 20. In the example shown, there are three index tabs 20, labelled "Default", "News Items" and "Hotels". The leaf shown is the "News Items" leaf and the "News Items" index tab 21 is shown highlighted. At the top right of the screen is a wastebin icon 22 which allows the user to remove a leaf and sent it to the wastebin.

There now follows a description of the interface whereby the web user can save a part or the whole of a web page.

The client interface allows the web user to save an element of a web page, or a link to the whole web page, to the repository; to follow the element's link immediately; E-mail the element to someone else; and/or open the repository.

Different set-ups can be configured for different situations. The interface allows the following options for saving an element:

The element may be stored in a specified part of the repository such as personal, private-shared, pooled or public;

The element may be categorised in one or more customised classifications as opposed to the default classification; and

The element may be described using one or more different types of identification such as customised name, text of link, title of page, visual representation (including the image portion of the element). Thus, the client interface permits elements to be saved accordingly to a defined degree of access, according to a defined categorisation and according to a defined description.

Different types of client interface can be used for different situations and it is likely that more than one may be available to the user in a given situation. Some interfaces are only available to the user if the web publisher has enabled them on their site, while other interfaces are always available to the web user by virtue of the fact that they are registered system users. The following description refers to the implementation of an interface which does not require the web-publisher to activate the service, that is easy to use, but is limited to the newest web-browsers. This interface uses extensions to the context menu of the user's browser, accessed in Microsoft Windows by clicking the right mouse button when the mouse is over the relevant element or page background. In the example to be described it is assumed that the user has previously downloaded and incorporated the extension into their browser. Turning now to Figure 2, an example of the context menu is shown. The user has previously registered with the service and has incorporated the

relevant proprietary extensions to her browser. Whenever she wants to save an element of a page (or indeed the frame or page itself), she simply opens up the context menu by using the right mouse button and then selects the appropriate service option.

In Figure 2, the user has opened the homepage 30 of their Internet Service Provider. The context menu 32 is shown overlying the homepage. The context menu includes two extensions, add to Copyn 34 which adds an element to the repository, and launch Copyn 36, which opens the user's repository. Other options may be added and customised to the user's requirements. In the example shown in Figure 2, the context menu has been opened with the mouse pointer overlying the link about Euro 2000 tickets. It is important to understand that if the user selects the add to Copyn 34 extension it will be this HTML element or collection of elements which will be stored in the repository and not the entire homepage of the homepage URL.

When the user chooses either to add the element 34 or launch the repository 36, the application checks for the appropriate cookie that would provide the server with the username and password. If the cookie does not exist, then the user is asked to log-in to the service, or to register as a new user. A cookie is then saved on the user's machine that will identify her the next time she accesses the service. In both cases the Element is saved in the appropriate location in the repository, assuming it has not already been saved, and, if the user had selected the 'Launch Copyn' option 36 her default repository is opened in a new browser window. Using a single user account with cookies means that it is very easy for the user to set up Copyn for multiple browsers and machines, Thereby enabling the sharing of the service between the office and home, etc.

The Repository Interface will now be described.

The user can choose between a number of different customisable web-based interfaces, via which the saved elements can be viewed and manipulated. The two preferred interfaces are:

A free-form "scrapbook"-like representation shown in Figures 3 and 4, and a hierarchical tabular representation shown in Figure 11 and which will be referred to later.

The user can toggle from one representation to another and the simple, hierarchical tabular representation of Figure 11 is always available, for spring-cleaning purposes, for a quick overview of the contents of their repository, or for any other reason.

Referring now to Figures 3 and 4, the repository interface provides the user with a wide range of functionality, including categorisation on screen display, a variety of services and means for sharing and connecting with other users.

Figures 3 and 4 are screen shots of the repository interface as it is seen by a user. In this case the user is displaying the interface in the Microsoft Internet Explorer browser. The interface includes a default categorisation 40 and a series of custom categorisations 42 which are defined by the user. In this case the user has defined four categories entitled, News Items, Basingstoke, Jenny Photos and Humour. The default category may be viewed as an in-tray for new elements saved.

The user of the system may be provided with a number of default categories which can be changed, by renaming, deletion or addition of fresh categories.

Categories are hierarchical, that is, Cards can be placed in categories, sub-categories, sub-sub-categories, etc. a

single Card can be placed in many different categories or sub-categories at the same time.

A given categorisation of a given set of stored elements together with their attributes, such as position, size etc. referred to as a "view" of those Cards. Each category is represented by a 'Leaf'.

For example, imagine a set of "bookmarks" about individual restaurants, in which each bookmark has been categorised by the location, type of cuisine and price range of the associated restaurant. Then three views of the bookmarks can be set-up: a "location" view, a "type of cuisine" view and a "price range" view.

The On-screen display of the illustrative "scrapbook" interface represents any category (or sub-category) of elements on screen by the relevant set of cards displayed on the appropriate leaf. The lay-out of cards on a leaf is similar to the lay-out of items on a page in a scrapbook, and the cards may be moved around by the user within a leaf, like loose cuttings, using "drag-and-drop". The cards 'remember' their new positions. The user can move a card from one leaf to another (thus re-categorising it), or to a "rubbish-bin" (thus deleting it), using "drag-and-drop". The user can 'resize' any card, with the card's contents being scaled or wrapped, accordingly, inside the card's border. Within the border of any card, the user can place their own comments, and/or other information which they select from a standard list of fields, such as date bookmarked, source page, etc. The user can toggle between different views of a given set of cards.

A number of services can also be provided. The user can upload and merge existing "bookmark/favorite" collections from their browser(s) into the repository at any time. This is particularly useful when a user first registers for the service. The bookmarks stored in the repository

can be clicked through just as they would be on the original referring page. One current exception is where clicking the link would execute a javascript program. The user is kept informed about bookmarked elements that have expired/gone stale, or whose content has changed.

Management information is available to the user, for example: listing those bookmarks which have not been clicked through for longer than a given length of time; or listing those bookmarks which are most often accessed.

The user can send any one or more of their bookmarked elements either individually or as a collection, to anyone else who has Internet access. This can be by email or as a message within the system. The sender can then categorise those particular bookmarks as having been e-mailed to that particular recipient; and both sender and recipient have the option of whether the sent bookmarks are linked or copied.

Various sharing and collaboration facilities are available. A user can create a "public" repository which, at the owner's option, any other registered user can read from or add to. This facility allows users to create different types of repository ranging from a "free-for-all" bulletin board to a "read-only" information site such as restaurant guide with links to restaurant web sites together with the repository owner's comments.

A user can authorise other, for example specially invited users, to have full access and use of a "pooled" repository. This service is particularly useful to clubs, societies, and the like where members share a common interest.

A user such as a school, university or corporation, can create a "private-shared" repository, for example running on their own web/database server, which enables students and/or staff to use the functionality of the system to

collaborate on web-based research activities. A variety of options are available giving different individual users different privileges such as read, write, modify, etc.

In the Figure 3 example, the leaf 40 is the default leaf which is shown highlighted. The leaf contains seven cards 44, 46, 48, 50, 52, 54 and 56 and the waste bin 46. The cards shown are selected to show examples of some of the different types of meaningful HTML elements which can be saved. Element 44 is an HTML DIV containing a link element, a DIV element divides a page into a number of logical sections. Here, an image has a brief description of the story and clicking on the image or the link will take the user to the linked web site as if they have clicked on the original web page.

Element 46 is a simple text link. Element 48 is a 2x2 table of advertisements. The bottom left and top right 58, 60 of which have links, identified by their bold borders.

Element 50 comprises text extracted from a linked news headline; the user chose to keep the text but drop the link. Element 52 is a banner advertisement in which an image is embedded in a link element.

Element 54 combines an image map and an image. The full map functionality is retained, for example, if the user clicks on the "Lawn and Patio" tab 62 they will be taken to that section of the amazon.com web site. Element 56 is also a DIV element comprising a link and some text, but which has been resized; the content has automatically obtained scrollbars to allow all of the content to be seen.

The user can move these seven cards around the screen, and resize them. The cards remember their size and location, so that when the user next returns to the

repository, the lay-out of the view is preserved from the previous visit.

Figure 4 shows a leaf from the News Item Category of Figure 3. It can be seen that the New Item Category comprises seven sub categories 64, identified as Asia, America, Africa, Europe, Sport, Angus Deayton and Local. Here the Europe sub-category 66 has been selected to display a leaf containing five cards 68. A waste bin 40 is also displayed in the leaf.

The manner in which the embodiments described operated will now be described.

An understanding of the relationship between the HTML and its DOM representation within the browser, and hence its availability to the browser scripting language, is essential to comprehend the manner of operation and will be described with reference to a simple example.

There are many subtle, and some significant, differences in the way that IE and NN turn the raw HTML of a web page into objects which can be accessed and modified by scripts, the DOM. However, the embodiments discussed rely almost exclusively on functionality common to both browsers, only deviating from this when a particular aspect of one browser or another offers significant implementation efficiency.

Figure 5 shows a simple web page comprised of some images and text. It is similar to the Card 40 shown in Figure 3. The first line ('This is my Table:') appears in a slightly larger font and although not visible in the drawing, in red. Below this text is a 2x2 table. The first column comprises 2 cells showing images, the second column includes images and text. Further subtleties can be seen in that the first row entries are aligned at the top of

the table cells and the bottom row entries are aligned along the bottom.

The raw HTML used by the browser to construct this page is as follows:

```
<HTML>
<HEAD>
        <TITLE>An HTML/DOM Illustration</TITLE>
</HEAD>
<BODY bgcolor="beige">
    <FONT color="darkred" size="+2">This is my
    table:</FONT><BR><BR>
<TABLE border="2" cellpadding="2" bordercolor="darkblue">
    <TR valign="top">
        <TD>
            <IMG SRC="/images/USWEST.gif" >
        </TD>
        <TD>
                <A href="/test.html"><IMG
                SRC="/images/etfront40" > Apricots are
                tasty</A>
        </TD>
    </TR>
    <TR valign="bottom">
        <TD>
            <A href="/experiment.html"><IMG
            SRC="/images/Strange"></A>
        </TD>
        <TD>
```

```
            Bananas are better! <IMG
            SRC="/images/USWEST.gif">
        </TD>
    </TR>
</TABLE>
</BODY>
</HTML>
```

Figure 6 is a summary of the DOM representation of the page. The picture only shows a small subset of the information available in the DOM about the content of the page. Specifically it only shows the "nodeType" (1=NODE_ELEMENT, 3=NODE_TEXT), "tagName", number of "childNodes", the non-default "attributes" of each node and the "nodeValue" of any text nodes.

It can be seen that the DOM representation mirrors the hierarchy of the raw HTML that was used to create the page. Each node has one parentNode and each element node can have zero, one or more childNodes.

The DOM representation of the page can be interrogated dynamically and, within constraints, can be modified without editing the underlying HTML. For example the position of elements on the screen can be changed by modifying some of their attributes, or the value of text strings changed. In the above example, if we changed the value of `document.getElementsByTagName("A")[0].childNodes[1].nodeValue` to "Oranges are tasty" our web-page would be modified onscreen such that it no longer told us that "Apricots are tasty" but that "Oranges are tasty".

Pages can be created on the fly, by a script manipulating the DOM directly without the need for any raw HTML, other

than the code of the script itself, being read by the browser.

There now follows a description of manner by which the user saves elements to the repository.

The operation of a user saving elements to the repository may be broken down into three main steps: setup and installation; finding the meaningful elements; and extracting the HTML for the meaningful elements found and returning it to the server.

The set up and installation requires customisation of the browser context menu and installation on a user machine.

The finding of the meaningful elements can be subdivided into the steps of: using the context menu as an interface with the users mouse over a node of interest; identifying a node supplied by the context menu; traversing the tree to look for collections of meaningful elements; finding related nodes if a given node requires a related node; and creating meaning where there is none.

The HTML extraction and return to the server can be subdivided into the steps of extracting the raw-HTML or DOM sub-tree from selected nodes; passing HTML data to a new window; selection by a user; and storage by the server.

These three main steps will now be described in turn.

SET UP AND INSTALLATION

To enable the customisation of the browser context menu, the following operations are necessary:

In Internet Explorer the user adds a new key in the windows registry under

HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\MenuExt\"My Menu Text"

Where "My Menu Text" is the text required for the new context menu entry.

The default value of the key is set to the URL of the page containing the script the developer wishes to execute if the user selects this menu entry.

The menu entry can be restricted only to appear in certain circumstances, for example only if the mouse is over an image. This is achieved by creating a binary value called Contexts under the key and setting its value accordingly.

In NN6, a new XUL overlay file, for example, navigatorCopynOverlay.xul is created which defines a new menu item as part of the context popup menu which can be referenced by setting the id of the <popup> element appropriately, namely <popup id="context">. An 'oncommand' value is attached to the menu item with the name of the script function to be called and the application is told where it can find the script via a <html:script> tag. Finally, the new overlay file is included in the global overlay file, in this case navigatorOverlay.xul, by adding the following line :

```
<?xul-overlay
href="chrome:[path]/navigatorCopynOverlay.xul?>
```

Optionally, submenu items can be added to the NN6 context menu and their appearance made conditional on the type of node which the mouse pointer was over when the context menu was activated.

Installation is relatively simple.

In order to extend the IE browser a small registry file is created which the user opens from the system web site. Doing so, having given the appropriate permission, will add the key to the users registry.

To install the extensions in NN6 requires the user to be presented with a signed script. A signed script is a normal script that has a digital signature that confirms the authenticity of the script. A signed script can request special privileges, not usually available to a browser script, such as the ability to modify the browser or access files on the user's system. If the user gives the script the appropriate permission, the modifications described above can be installed.

The step of finding the meaningful elements, and the various sub-steps will be described with reference to Figure 7.

To select an element to be added to the repositories, the user moves her mouse to that element and then activates the context menu over the item of interest. This is shown at step 100. Thus, the context menu is used as an interface with the user's mouse over the node of interest. The user can now select the add element option (34 in Fig. 2) to add an element to the repository. At step 102, a handle to the Node is returned to the script from the DOM over which the mouse was when the context menu appears. In IE this Node can be accessed from 'parentwin.event.srcElement' and in NN6 from 'document.popupNode'. These are both the same type in the DOM, an HTML Node. This Node will be referred to as 'myNode' for the purposes of the following.

Identification of Node supplied by Context Menu

At step 104, the script identifies the type of myNode (via myNode.nodeType). The options of interest in the HTML implementation are typically types 1 and 3. Type 1 is an ELEMENT_NODE which means that the node received is an HTML Element, and Type 3, which is a TEXT_NODE. Text nodes hold all the text data outside the HTML'<' and '>' tag brackets. Often text nodes are nothing more than the carriage returns between two lines in an HTML file but more interestingly this is where the text shown on the screen can be obtained from the DOM. In the DOM representation of Figure 6 a large number of TEXT NODES consisting of carriage returns and white space were omitted for simplicity.

Element nodes can be further distinguished by their tagNames, as can be seen from Figure 6. Different useful data can be obtained from each tag type. For example the source of an image file can be obtained from the 'SRC' attribute of an <IMG> tag or the row and column data from the childNodes of a <TABLE> tag.

At step 106, myNode is examined to determine whether it is a meaningful element according to the defined rules. If it is, at step 108 the element is added to the list of meaningful elements.

The script now traverses up and down the Node tree, looking for meaningful collections of elements by looking for meaningful ancestors and descendants. For example from a link (<A>) the script looks at all the childNodes, and their childNodes and so on to search for text nodes or image tags that form part of the link. The script then looks up at the parentNode, and its parentNode etc. until it reaches the document <BODY> which is the highest level node that could be of interest in this context, noting on the way if the link is part of a <TABLE>, <FORM>, <DIV>,

<SPAN> node etc., each of which could represent the common ancestor of a meaningful collection of elements.

In Figure 7, at step 110 the process first looks for childNodes. If there are, the handle of each childNode is in turn passed to the script at step 112 and steps 102 to 110 are repeated for each childNode in turn. The process at step 114 then looks to see whether the parentElement of the current element is the BODY element. If it is not, at step 116, the handle of the parent element is passed to the script and steps 102 to 114 are repeated. If the answer at step 114 is yes, the process asks whether it is policy to capture BODY elements at step 118. If yes, the BODY element is added to the list of meaningful elements at step 120. In any event, the script is now ended at step 122.

Looking at this process in more detail, and referring to Figure 8, consider the example HTML page and the DOM at Figure 6. If the user activates the context menu over the image or text in the top right hand cell of the table, myNode will refer to the Node second from the left in the penultimate row of the diagram shade node 130. This is an Element Node representing an anchor tag ('<A>') and its descendants represent a meaningful collection element so this node must be noted. The Node tree is now traversed looking for meaningful descendants and ancestors.

First, the childNodes of myNode are located at and 2 Nodes 132, 334 are obtained, shown shaded in Figure 8. These nodes are Element Node 132 for an <IMG>, another meaningful element to be noted, and a Text Node 134 stating that 'Apricots are tasty' which is another meaningful element, despite the fact that technically this Node is not an element. The manner is which this type of Node is dealt with will be discussed later. Again, this

element is noted. Three meaningful elements are now captured.

The search is then reversed and the parentNode 136 of myNode looked at. This is an Element Node for a Table Data ('<TD>') tag representing a single cell in our table. For the time being this is considered not to be a meaningful element as will be discussed. This Node's parentNode 138 is then examined to obtain an Element Node 138 for a Table Row ('<TR>') tag. Again this is not considered to be a meaningful element.

The next parentNode 140 is examined to obtain an Element Node for the Table ('<TABLE>') tag that represents the whole of our 2x2 table. This represents a meaningful collection of elements, the whole table, and is noted. The parentNode of the TABLE is the BODY 142 of the whole document which again represents a meaningful collection of elements and also a stopping point for our Node traversal.

Capturing the body of the page as represented by the BODY element is different to bookmarking the location of the page. For example, the first page of a newspaper will change from day to day and so a user who wishes to capture the front page on a special occasion will actually need to capture the body of the document as opposed to the URL of the page.

In practice this Element and its descendants may not be captured as the amount of data involved may be quite large. If it is decided to capture it then it cannot be saved 'as-is' and its content must be put into a <DIV> Element which can be stored and retrieved from the database and displayed within the confines of another document. The manner in which a node is handled will again be discussed later. DIV and SPAN elements can be used to create freely positional "sub-pages". The content in a DIV or SPAN element can be set to move with its parent

Element, hidden or made visible and even occasionally resized in proportion to the DIV or SPAN element.

A rule set is used to determine and identify 'meaningful' Nodes, the decisions used for when to stop searching up or down and special treatment of Nodes, such as for the Body Element above. This rule set is based on the DTD for HTML with as little overruling as possible - this means that keeping the system up to date is more straightforward as the specification of HTML changes, and also provides an approach to generalising the technique described to other markup languages that come with their own DTDs.

For some types of nodes the script must also find associated or related nodes or data. A second set of rules is used to facilitate this. For example if a user activates the context menu over an image map ('<MAP>') the script must find the image that uses the map; the collection of images in the document can be obtained from the array of image Nodes held in 'document.images' within the DOM. MAP elements can also be applied to OBJECT and INPUT elements. These must also be searched to find the appropriate element to be matched to the MAP. It is then a simple matter to scan through these to find the images, objects and inputs using an image map and in particular the one using the image map on which the mouse was placed. In another situation style sheets/style definitions may be needed to interpret the class attributes of nodes. This may be done in one of two ways: the script could locate and load the appropriate style sheets and cssRules or the script could record the non-default style settings of the node itself. It is preferred to extract the style information of each node independently but this is not essential.

Alternatively, global style settings can be captured by a straightforward DOM function call.

In some cases, non-meaningful elements need special treatment to make them meaningful.

Earlier it was stated that '<TD>' and '<TR>' tags did not represent meaningful collections of elements. In isolation they do not - without a '<TABLE>' tag - represent well formed HTML. To the user, however, it is appealing to select rows from tables or groups of adjacent cells. It is made possible to select combinations of nodes which share a common ancestor node type. For example, table data or table rows can be lifted from the table. In this situation the script would create a new ancestor of the appropriate type, possibly using the formatting attributes of the actual table from which they are being selectively extracted. A third set of rules is used to facilitate this which will be referred to later.

A list of the meaningful Elements and common ancestors of meaningful collections of Elements has now been obtained.

The third stage of the process is to extract the HTML for these meaningful Elements and Returning it to the server.

Having drawn up a list of meaningful Elements, or collections of Elements, the script now extracts the required data from the DOM for each of them in turn. This data will then be passed to a new window before being sent to the server. This process is illustrated in Figure 9.

There is a choice between extracting the raw-HTML, or the DOM sub-tree from Selected Nodes.

The HTML represented by the Elements and their descendants can be recreated or copies of the relevant sub-trees of the DOM itself copied. The choice in practice depends on the performance of the different browsers at the extraction of the data or copying the DOM subtrees.

If the implied raw HTML is created, a number of techniques may be used. It must be noted that this HTML may have been created by a script on the publishers web site and may not represent the actual HTML passed from the web site's server. Alternative approaches will be described later.

Referring back to Figure 8 and commencing at node 130 which relates to a link containing an image and the text 'Apricots are tasty'. The whole of the process must be repeated for each meaningful Element in the list.

Referring to Figure 9, a blank string "myHTML" is created at step 150. At step 152 a check is made whether the element is of the type ELEMENT_NODE. If not, a check is made at step 154 to determine whether the element is of the type TEXT_NODE. If, at the step 152 the element is determined to be an ELEMENT_NODE, at step 156 the opening tag ("<A ", in the example being considered) from the tagName of the Node (myNode) is added and a list of the attributes checked for the Element from 'myNode.attributes' and for any that have non-blank values add them to the myHTML string. In the example, myHTML now reads "<A href='/test.html'". The same exercise is repeated for any style settings that have non-default values by scanning through the 'myNode.style' array. In the example there are no style settings so myHTML is unchanged. The opening tag (myHTML="<A href='/test.html'>") is then closed. Thus, in Figure 9 step 156 is executed in the order of the opening HTML <and name tag, non-blank attributes, non-blank style settings and finally the closing angle bracket>. In IE the list of attributes is very long and goes well beyond the list of attributes specified in DOM2. The list is thus restricted to the list of attributes applicable to each Element type - this can be obtained from the DTD. For the sake of efficiency the search through the style setting may be

restricted to the core values relating to size, position and colours.

We now recursively repeat the exercise for each childNode, and in turn for each of their childNodes - including non-meaningful Elements - and their childNodes etc. This is shown at step 158 in Figure 9 at which it is determined whether there are any childNodes. If there are, at step 160, the handle of each childNode is passed in turn to the script and the process is repeated recursively for each childNode. The result is then appended to my HTML. Referring to the Figure 8 example, the first node encountered is the IMG element. Repeating the above exercise of extracting attributes and styles, myHTML="<IMG src='/image/etfront'>" is created. This node has no childNodes and so a check is made to see if a end-tag is appropriate for this type of Element. In this case it is not, as, according to the DTD for HTML, <IMG> elements do not have end-tags so the local myHTML is returned back to the parent node. For the link node, myHTML now reads ="<A href='/test.html'><IMG src='/image/etfront'>". In Figure 9, the step of looking for an end tag is shown at step 162. If present, the end tag is applied to myHTML at step 164. If not present, or after application of the endtag, the finished script is returned to myHTML at step 166.

The next childNode of the link is a text Node from which is extracted the nodeValue which is returned to the parentNode. For the example link node, myHTML now reads ="<A href='/test.html'><IMG src='/image/etfront'>Apricots are tasty". There are no more childNodes so an end-tag is added to myHTML, if appropriate for this type of Element, to get the final result of

myHTML="<A href='/test.html'><IMG src='/image/etfront'>Apricots are tasty</A>"

The process is summarised by the following pseudo code.

```
Function extractHTML(myNode){
   create empty string myHTML=""
   if (myNode is an Element Node (i.e.
myNode.nodeType==1)) do {
      myHTML = myHTML+"<"+myNode.tagName
      for each member of myNode.attributes do {
         If specific attribute is non-default myHTML =
myHTML + " [attribute name]=[attribute value]" or
[attribute name] for boolean attributes.
      }
      if (any member of  myNode.style is non-default)
myHTML = myHTML + "STYLE=' "
      for each member of myNode.style do {
         If specific style is non-default myHTML = myHTML +
"[style name]:[style value];"
      }
      if (any member of myNode.style is non-default)
myHTML = myHTML + " ' "
      myHTML = myHTML+">"
      if (number of childNodes (i.e.
myNode.childNodes.length) > 0) do{
         for each member of myNode.childNodes do {
            myHTML = myHTML + extractHTML(childNode of
myNode)
         }
      }
      if the tagName of  myNode requires closing tag
   myHTML = myHTML+ "</"+myNode.tagName+">"
   }
      else if (myNode is Text Node (i.e.
myNode.nodeType==1)) do {
      myHTML = myHTML + myNode.nodeValue
   }
   return myHTML;
}
```

This is represented by Figure 9.

This description has glossed over one essential task the script must perform on the extracted HTML (or DOM subtree) before it is passed to the new window. Many websites reference images and links etc. relative to a base URI, often the domain of the page being viewed. In the example the images SRC attribute looks like the following SRC='/image/{filename}' - this reference is relative to the domain of the publisher's server. If the user attempted to display this image from the repository site he would not see the image as the repository will not have a copy of the image file. What the script therefore does is replace SRC='/image/{filename}' with SRC='http://{domain_name}/image/{filename}'. This is easily done as the DOM subtree is traversed. Each time an attribute is found that may need changing, such as 'SRC' for <IMG>, 'HREF' for <A>, a few string operations are performed that convért the relative URI to an absolute URI. A full list of attributes whose values are URI's can be obtained from the DTD. The process that must be executed to convert relative to absolute URI's must satisfy the following Request for Comment rfc 1808 which can be found at www.ietf.org/rfc/rfc1808.txt. If the base URI in this example was 'www.domain.com' the final HTML to be captured would then read

myHTML=
"<A href='http://www.domain.com/test.html'><IMG
src=http://'www.domain.com/image/etfront'>Apricots are
tasty</A>"

Instead of

myHTML="<A href='/test.html'><IMG
src='/image/etfront'>Apricots are tasty</A>"

There is now a list of meaningful elements or the common ancestor that makes a collection of Elements meaningful, together with the HTML that represents each of them (and their descendents) in the DOM.

Capturing the Javascript associated with an "HREF" or "event" is theoretically possible but may cause unpredictable behaviour. The scripts in a page can be obtained from an array of script elements from the DOM. This array could be recreated in the HTML being saved, thereby ensuring that the script attached to the "HREF" or "event" is available when the repository displays the saved element. Variable and function names in these scripts may clash with names from other sites and may well refer to elements on the original web site that are no longer available once the element has been saved out of context. The ability to save the scripts associated with element attributes (including mouse and keyboard events) may therefore be disabled.

The HTML data is then passed to a new window (or a new layer on the same page). The script, having identified the Nodes representing the common ancestor of each meaningful collection of elements, or having created a virtual ancestor where such a node does not exist, takes the HTML represented by each Node and its descendants and passes it as an array of data to a new window it creates. The HTML passed to the new window is written into a series of layers, or '<DIV>' elements all of which are hidden from view apart from the default option, which is the HTML corresponding to the actual element over which the context menu was activated.

In its simplest manifestation the layers are created by the following type of script (in pseudo code):

```
for (i=1 to number of meaningful elements) do {
```

write the following HTML to our new window

"<DIV ID='myLayer[i]' STYLE='visibility:hidden'>
myHTMLArray[i] </DIV>"

}

If our default option was element no. 2 (for example) we would then modify the style as follows:

document.getElementById('myLayer2').style.visibility='visible'

The User Then Makes His Selection. On this new window is a FORM, with a pulldown menu of options, a <SELECT> tag, corresponding to each of the meaningful collection of elements passed from the main window. As the user chooses different options from the menu the corresponding layer is made visible and the others hidden. This is done by switching the style visibility setting of the DIV to 'visible' and 'hidden' accordingly.

This is illustrated in Figure 10 which shows a screen shot of a Window 200 in which the selected area to be saved 202 is displayed. The user selects from a drop down menu 204 what he or she wants to save, for example the entire table, an image or a link and clicks the "add to Copyn" button 206 to save the selection to the repository. A reset button 208 is provided to enable a selection to be cancelled.

When the user has finalised his choice (in our example between the text, 'Apricots are tasty', the image 'Love a Book', the link, which includes the text, the image and a target for the link, and the whole 2x2 table) he clicks on a button to 'post' the results from the form to a web server program (for example a cgi script written in Perl) running on the repository server. Posting is one of the methods of returning data to the server from an HTML form.

Until now there has been no interaction with the server. Only the selected HTML is passed, together with other useful pieces of information such as the URL of the page from which it was obtained, the size of any image files (only possible in IE at present) etc. The exact choice of data to be returned will depend on customer demand but this data is generally obtained by a limited number of methods including the following:

Extracting HTML for selected elements on the page; the Height and Width of the element as currently rendered by the browser (this is obtained from the offsetHeight and offsetWidth fields) which is useful for determining the size of the element for display on the repository; Obtaining browser or system data from data made available from the DOM (e.g. type of browser or operating system); Information about the web site and domain (such as the URL of the page); and Date and Time data.

The server then stores the data as follows. The server script first checks for a 'username' cookie. If it does not find one the user is invited to log-in or register. The user details are confirmed with, or stored in, a database table on the server. This use of cookies for identifying users and validation of passwords etc. is common practice online and will not be described any further.

Once the user has been validated, the server script takes the data provided by the form and adds it to the user's repository. An SQL query may be made to ensure that the data is not a repeat of content already in the users repository.

The data is stored in the 'default' category determined by the user's predefined preferences.

Once all this has been done, the content of the 'new window' is replaced with a message from the server. A confirmation message, showing what has been saved, is displayed in the new window. After a short preset period of time, for example 5 seconds, the new window closes itself.

The HTML representing the user's selected generic Element has now been passed to his repository for subsequent retrieval.

Database Implementation

The following representation of the database and its associated tables and data allows the invention to be recreated but may not necessarily the most efficient implementation which could be developed. Sufficient information about the requirements is, however, provided to allow a more sophisticated database to be developed.

The information set out below relates only to the implementation of the invention and not to other data and services that may be useful from a commercial point of view. For example, in a commercial implementation we may seek further user data beyond the Name and Password (e.g. e-mail address etc.). Implementation of such additional features is straightforward for one of ordinary skill in the art.

The core data will be split into 9 data tables (more tables may be added later depending on business requirements). Taking each data table in turn, the purpose of each table and the primary fields required is as follows:

User Data Table

This captures information about each user and basic preference data such as their default group and default repository.

User Name	Name by which user identifies himself
User Password	Password user selects to control access to account
Default Group = Groups Data {Unique Id.}	Default collaborative Group to which user belongs (may be blank)
Default Repository = Repository Data {Unique Id.}	Default repository of saved elements - each user has one or more repositories.
Unique Id.	System generated identifier for User.

User Data Table

Element Data Table

This is the core data saved by the client interface described. It holds the HTML, domain details etc. but nothing about how this data is to be displayed on the repository interface.

Raw HTML	HTML extracted and saved by Client Interface
Source Domain Name	Domain name of site from which the raw HTML was taken
Source Page URL	URL of the web site from which the raw HTML was taken
Date/Time Created	Date/Time the element was saved
Date/Time last visited	Date/Time the element was last clicked on (if a link)
Owner Repository = Repository Data {Unique Id.}	Repository within which the element is saved. Elements can exist within more than one View for the same repository.
Copy of Me = Element Data {Unique Id.}	Location of a copy of the element (created if user sends copy of part of repository to another user for example). This copy of the Element may need updating if the underlying element changes. This copy can have copies of its own, etc.
Unique Id.	System generated identifier for Element.

Element Data Table

Card Data Table

The information in this table captures information about the display, formatting and position of the Element Data. The card has information about which leaf it is displayed on. Any given Element can be associated with several different Cards.

Associated Element = Element Data (Unique Id.)	Location of element to be displayed on this card
Position/Size etc.	Examples of customisation options specific to each card such as location on screen (within the leaf)
Background Colour etc.	Examples of customisation options that can be common to many cards - these can be overwritten by, or inherited from, Owner Leaf.
Comment/Description Text etc.	Examples of text fields the user can add, or modify to describe or comment on the card/element.
Owner Leaf = Leaf Data (Unique Id.)	Identifier for the Leaf of which this card is a part.
Date/Time last visited	Date/Time the element was last clicked on (if a link) - specific to this card.
Unique Id.	System generated identifier for Card.

Card Data Table

Leaf Data Table

The User's screen, in a given view, is split into a number of Leaves navigable by tabs, similar to a spreadsheet in MS Excel and other products. Each Leaf holds information about its own display as well as default values for any Cards placed in it. In essence Leaves can be used to categorise and classify Cards and hence Elements.

Owner View = View Data {Unique Id.}	The View of which this Leaf is a part.
Leaf Title	This a descriptive title used for tab label.
Reference to View = View Data {Unique ID/}	In order to accommodate sub-Leaves a leaf can include a pointer to a View - this View and its Leaves will appear within this Leaf (see Figure 4 for illustration).
Background colour, text font, border type etc.	Customisation options for the leaf that drive its display. Some settings may be inherited from default values at View level.
Background colour, text font, border type etc.	Default settings for customisation settings for cards that appear within it.
Unique Id.	System generated identifier for Leaf

Leaf Data Table

View Data Table

A View is made up of a collection of Leaves and hence cards and in turn Elements. Overall View settings can easily be copied from one Repository to another.

View Name (descriptive)	This a descriptive title used by the User to identify the view (we may also have a short form title for use on menu options)
Owner Repository = Repository_Data {Unique Id.}	This is the Repository to which this view applies.
Overall customisation data (e.g. page size, type of waste-bin etc.)	Some customisation data exists at View level - this includes location of waste-bin, position of leaf tabs, default values for Leaf settings.
Unique Id.	System generated identifier for View

View Data Table

Repository Data Table

Each user or collaborative Group of Users has one or more repositories of data. The identification and administrative data is held in this table together with the default View associated with the Repository.

Owner User/Group= User Data {Unique Id.}, Group Data {Unique Id.}	Each Repository has an owner/administrator responsible for it. This can be a single User or a Group.
Default View = View Data {Unique Id.}	Each Repository has a default View.
Unique Id.	System generated identifier for Repository

Repository Data Table

Groups Data Table

Users can belong to collaborative Groups that can access shared repositories - this captures information identifying the Group and its default Repository. Universal groups allow users to make their Repositories/Views available to everyone, e.g. for public read access.

Group Name (descriptive)	Descriptive title for the Group (may also have shorter version for menu labels)
Owner User = User Data { Unique Id.}	Administrator/Owner for the Group - this User is responsible for Repositories (and hence Views etc.) owned by the Group.
Default Repository = Repository Data { Unique Id.}	Default Repository for the Group
Unique Id.	System generated identifier for Group

Groups Data Table

UserGroup Data Table

This table maps Users to Groups. It is used to determine which Users are members of which Groups.

User = User Data {Unique Id.}	Name of User belonging to Group
Group = Groups Data {Unique Id.}	Group identifier
Unique Id.	System generated identifier for UserGroup linkage

UserGroup Data Table

Permissions Data Table

This table is used to restrict and manage access privilege to various data in other tables. For example it can be used to limit access to a Repository or view.

Associated Data = {Unique Id.}	Unique Id. From any of the following above data tables : Element, Card, Leaf, View, Repository, or Group.
Associated Table = Table Name/Data Type	Name of data table to the which the above identifier refers.
Recipient User/Group = {Unique Id.}	User or Group to which this permission relates
Grantor = {Unique Id.}	User or Group that owns this permission.
Type of Permission	Whether the permission relates to ability to read, modify, create, delete, administer etc.
Unique Id.	System generated identifier for Permission

Permissions Data Table

The Permissions data table is very important. The data can be used as follows:

A Group owner may grant the right to administer Group membership to another User. In this case the Group owner is the Permission Grantor, the second member is the Recipient User, the Type of Permission is administration, the Associated Data Table is the Group data table and Associated Data is the Group to which the second user is being given the permission.

A User may grant universal read access to a specific View of a specific Repository. In this case the Permission is set for the View - the Grantor is the User, the Type of Permission is read access, the Recipient Group is the Universal Group and the Associated Data is the View. A Permission of the Repository is created with the same settings. The repository cannot be 'looked' at other than

via a View and so granting this Repository Permission does not allow access to other views.

A Group may choose to organise itself with each User having full access to one Leaf each and read access to all the other Leaves. This can easily be achieved by setting the appropriate permissions on each Leaf.

The database also stores a copy of the various DTDs used to define the syntax of HTML markup constructs. These will be the first of many DTDs to be captured in the database and will form the dataset from which the rulesets, required to capture and display broader XML elements, can be developed and recorded.

The database used may be a standard SQL database or other type of relational database, which the web-server accesses via Perl/CGI, or another interface mechanism between the web server and the database.

This data structure set out above allows groups, views, leaves, cards, permissions etc. to be customised.

The repository user interface will now be described in greater detail.

There are two aspects to the Repository User Interface, ("RUI") the representation of the data in a relational database as described and the Free-form visual user interface, which is one implementation described.

Before describing the mechanics of how the visual interface works it is useful to give a brief description of how the database structure ties in to the practical use of the system:

"Users" can belong any number of collaborative "Groups" (including none).

The administrator of a group manages the repository access privilege of group members and the administrator can also allow universal read access to a repository.

Users and Groups can have one or more Repositories. Repositories can have more than one View. The user can switch views at any time by choosing the desired view from a drop down menu.

Views are constructed of a customisable set of Leaves. The number of Leaves can vary, as well as their layout on the screen. In the default layout, the Leaves overlap each other with non-overlapping tabs at the top to allow the user to switch from leaf to leaf. Leaves can have different background colours or images. Leaves provide default customisation parameters to the Cards displayed on them. A Leaf tab can point to a View to be displayed completely within the Leaf to form a type of sub-Leaf. This allows the type of multi-level leaf structure illustrated in Figure 4.

Leaves display a number of customisable Cards. Each card can be customised or can inherit its settings from the default values stored at Leaf level. Customisation includes background colour, including transparent or even a background image, border type, whether a comment field should be displayed etc. Each card displays one Element and can have comments/descriptions attached, which can include hyperlinks added by the user. Cards can display information about the page from which the Element was stored, date of last access etc. The card can be repositioned on the screen and resized by dragging the mouse. The card can be moved (or copied) to another Leaf by dragging it onto the new leaf tab. The card can be removed from the view entirely by dropping it onto the

waste bin icon. Changes in customisation settings are returned to the server so that the View is kept up to date.

Each Element represents the ancestor Node of a meaningful collection of Elements stored from a web-site via the Client Interface described earlier. This is rendered by the users web-browser to appear within the card with the customisation set as required by the user.

The previous description described the data structure underlying the invention in some detail. This section sets out how this is tied in with the user interface. Rather than describing the interface sequentially, as was done for the Client Interface, this section will describe how all the key functionality is achieved.

Overall Structure of the Repository Interface.

The user accesses a repository by opening their home-page on the server. This site can also be launched by using an extension to the browser context menu, as described earlier.

The data sent to the user's web browser from the respository server consists of 3 main groups:

1. Javascript Code (browser side script)

A fairly substantial piece of Javascript will be delivered to the web browser. This would typically be cached automatically by the user's machine and so there will be very limited performance overhead. Much of the customisation data specific to the Repository/View combination being viewed will be passed to the script as parameters which the script uses to build the page being viewed, customised for the situation.

The way that the script works and how it obtains, processes and updates the customisation data will described in some depth later.

2. Database dependent HTML generated by a CGI/Perl script (server side script).

It is preferred to implement the web-server scripting and database access using CGI/Perl but this is not the only choice available. The way that this code works for the significant parts of the process will be described in some detail later. The process will be similar regardless of language choice on the server.

3. Static HTML. Very little of the RUI is static HTML. Most of it is customised for the specific user/repository/view - either by the web-server or by Javascript.

Obtained Data.

User Details

The repository site reads a cookie, containing a username and encrypted password combination, specific to the repository server's domain when the user first requests access to the repository. This is checked against the values stored in the User data table, using a simple SQL query. If there is no cookie stored or the username/password combination is invalid the user is requested to try again or to register to the service. This whole mechanism is commonplace on the Internet and so will not be described in more detail.

Default Settings

Once the user has been validated access can be had to all their preference data from the User data table. This includes their default Repository and Group - this data is

used to determine the initial data/display they see on the RUI (i.e. their repository home page).

The default Repository is looked up in the Repository data table. This then provides the server based script with the default View, with its customisation data. This in turn is used to find all the Leaves included in this View, with their customisation data. These in turn give the cards with customisation data and finally the Elements themselves. This data is obtained by a number of database queries.

A significant block of HTML data; customisation settings pertaining to the User's default Repository and its default settings have now been extracted from the database.

There now follows a description of how the data from the database is delivered to the browser script.

There are a number of ways in which this can be achieved but they involve the same basic principal. The following describes a specific solution utilising the IFRAME element, the HTML code element for creating floating frames.

The browser side script creates a hidden IFRAME element on the page, it is hidden by setting its style parameter accordingly, which receives the data from the server script by setting the IFRAME's SRC attribute to call a server side script.

The following type of command would achieve this:

```
document.writeln("<IFRAME NAME='hdn1'
SRC='/perl/myData.cgi'
STYLE='visibility:hidden'></IFRAME>");
```

During the construction phase of the web page this allows the server-side script 'myData.cgi' to be executed. This server side script in turn creates a new browser side script, within the hidden IFRAME, containing the customisation data we require. This is done by making the database queries mentioned in the previous section, and writing the results out into a series of arrays.

These arrays allow the data to reflect the hierarchy of items to be displayed. Each piece of element data is stored within a card data array, together with customisation data. The data for a group of cards is held in a leaf data array, the leaf data is held within a view array.

Once the script (myData.cgi in this case) has finished executing and the results fully loaded into the IFRAME, this data is available to the main browser script that is controlling the creation of the page. The content of the IFRAME can be accessed via :

document.frames.hdn1.arrayvariablename etc.

Using the customisation data from the database.

The overall structure of the page is determined, either by HTML received from the server or by the script. This process is very commonplace and will not be described here. At this stage there is a fairly content free page, perhaps displaying a logo, copyright and terms and conditions statement etc.

Once the customisation data has been loaded from the server the controlling script proceeds to create the remainder of the web-page. The overall customisation data is used to add a little more detail to the page for example the choice of wastebin image and by changing the

default colour scheme. This is done by modifying the style settings of items that already exist within the DOM and inserting new items, such as the wastebin (the wastebin is added in much the same way as Leaves and Cards which are described below).

The required number of Leaves is added, the visibility setting of the default Leaf being set to 'visible' and the others to 'hidden'. On each Leaf the Cards are drawn.

Leaf construction and manipulation

Leaves will be added and deleted by the user after the page has finished loading. Therefore, when first inserting the leaves into the document, the same mechanism can be used. The DOM2 provides a standard way for doing this, and the two browsers (IE5+ and NN6+) provide a convenient, but non-standard, mechanism for inserting it into the document. These methods themselves do not form part of the DOM2 specifications but are more efficient than the DOM2 methodology.

In both cases a blank string (myHTML, say) is created. The script loops over the number of Leaves, incrementally adding HTML as text to myHTML. For each Leaf we do something like the following:

```
myHTML=myHTML+"<DIV ID='Leafn' STYLE='leafstylen'></DIV>"
```

Where Leafn is an identifier for Leaf number 'n' and leafstylen incorporates the customised display settings for the Leaf, making sure that the Leaf Style takes note of which Leaf is to be displayed initially.

For NN6 now take myHTML and create a DocumentFragment (a free standing DOM subtree) from it using the createContextualFragment method of the Range Element and

insert it as a new child of the BODY element using the appendChild method. Note that the same result could be achieved by creating the Element and its attributes one at a time by using DOM2 compliant methods. Whilst this is a purer approach it is far less efficient.

For IE5 take myHTML and use the insertAdjacentHTML method of the Body Element to insert the HTML before the end of the Element.

Small 'tabs' are created to appear at the top of each layer. These are created using the same layer technology as the Leaves themselves with the DIV elements structured to be appropriately dimensioned and placed just above the Leaves themselves. On each DIV element is placed a text based link. The text of the link is the Leaf Title, from the customisation data, and the HREF attribute is set to run a simple javascript function that switches the Leaf being displayed to the one corresponding to the tab being clicked on by the mouse. It is possible to use a mouse event to trigger the leaf switch in place of the HREF approach for more refined handling. The script merely switches the visibility style flag on each Leaf layer to achieve this. Additionally when a user selects a tab its background colour is changed (using its style setting again) to highlight the active Leaf title.

Sub-leaves can be created within the layer representing the leaf, with tabs appearing at the top of the sub-Leaf, immediately below the tabs for the main Leaves themselves. This is achieved by using a Leaf Tab as a pointer to another View which is then created within the Leaf (as opposed to within the BODY of the document). In the above description of creating a Leaf the appendChild (or insertAdjacentHTML) method is applied to the Leafn element instead of the BODY element.

At any point the user can insert a new Leaf by running a script function, which can be attached to a button, a main menu item or the context menu. This script creates a new empty leaf using the same technique as described for creating the other Leaves. In this case there is no data to be obtained from the database so the new leaf settings are set to the default levels for the View until they are overwritten by the user.

The overall page structure is now set up and the Leaves are displayed. But they have no content.

Card construction.

Cards are constructed in a similar way to the Leaves. In this case, however, the card is a more complex item to construct.

A card has a few core parts:
The containing layer, which is the containing outer boundary of the card; the element layer, a sub layer of the containing layer that contains the Element stored in the database; the comment layer, a sub layer of the containing layer that contains any comments and additional text fields related to the Element stored in the database; and the resizing layer, a sub layer of the containing layer that provides a box that the mouse pointer can click on to resize the containing layer and with it the element and comment sub-layers.

These layers are called cardLayern, cardSubLayern, cardCmtLayern, cardRszLayern in the following description, where n refers to the card number and is unique within the View. In other words the numbering system does not restart with each Leaf. The customisation settings, passed from the database via the IFRAME element, are captured as STYLE settings associated with each layer that makes up the card

(cardLayerStylen=cardLayern.style, cardSubLayerStylen, cardCmtLayerStylen, cardRszLayerStylen).

For each card, a piece of HTML (say 'myHTML') is constructed along the following lines:

```
myHTML=myHTML
+ "<DIV ID='cardLayern' STYLE='cardLayerStylen'>
+ "      <DIV ID='cardSubLayern'
STYLE='cardSubLayerStylen'>"+myElementData+"</DIV>"
+ "      <DIV ID='cardCmtLayern'
STYLE='cardCmtLayerStylen'>"+myCommentData+"</DIV>"
+ "      <DIV ID='cardRsvLayern'
STYLE='cardRszLayerStylen'></DIV>"
+"</DIV>"
```

Where myElementData is the raw HTML captured by the user and obtained from the database and mycommentData contains the comments and descriptors that the user has opted to display.

This piece of HTML is then inserted into the appropriate Leaf Layer (as opposed to the BODY Element).

Since the creation of the cards will cause their associated Elements to be loaded from their relevant third party servers (as determined by the SRC attributes of images etc.) the order in which they are loaded needs to be controlled. The script staggers the creation of cards on all but the default leaf, in order to allow time for the cards on the default leaf to be loaded. This delay is overruled if the user switches the display to another Leaf. This extra sophistication is built into the leaf switching script attached to each tab (as described in the previous section). A flag is checked to see if the cards on the new Leaf had been created, if not, then the cards are created immediately.

The position style setting of each layer is set to 'absolute' and then to define the dimensions as percentages of the containing layer (cardLayern). This means that the layers will all move and resize together.

Control of Card Content.

Stored elements and meaningful collections of elements are being displayed out of the context in which they were created and they may not be displayed the intended way.

Some elements provide their dimensions as a matter of course, as is the case for most images for example or where the original web publisher required for a specific layout. In addition, the actual height and width of the element as displayed on the screen was captured when the user saved the element originally.

This information is used to determine the size and shape of the element, as it should appear in its card, and clip the region to ensure that the elements do not spill out over the edge of the containing layers. This can be done setting the clip style setting for the cardSubLayer.

For some Elements, in particular images - with or without associated link, the dimensions of the Element can be set to resize with the dimensions of the cardSubLayer. This is done by setting their position style to 'absolute' and fixing their width and height to fixed percentages of the cardSubLayer. This has the effect of causing the image to change shape as the user changes the shape of its container. This will be possible for other select Elements. For other Elements if the cardSubLayer gets too small to contain the Element then the content will be clipped or scroll bars will appear (depending on the Element type). The scroll bars appear if the overflow style setting of the cardSubLayer is set to 'auto'.

Moving and Resizing Cards, moving cards to another Leaf or dropping in the Wastebin.

With both IE5 and NN6 browsers mouse events can be attached to various elements, including the DIV elements from which the card is built.

The mouse events of interest are : onmousedown; onmousemove; and onmouseup.

Many articles have been written about moving items on web displays using the mouse and so a broad overview only of one way of doing this is given Further information may be found at http://developer.netscape.com/viewsource/goodman_drag/goodman_drag.html

onmousedown

Once the cards have been created the onmousedown method of each cardLayern is assigned to a script function ('engageLayer'). This function now 'listens' for this event being triggered by the user's mouse interacting with this element on the screen. This function will be called when the user presses down a mouse button on the portion of the layer not covered by other items and not if the mouse button is not pressed down. When it is called this function sets a global variable ('selectedLayer') equal to the element returned by the event (NN6=evt.target, and IE=window.event.srcElement), records the (x,y) co-ordinates of the mouse when is was pressed down and sets the onmousemove method of the document equal to a script function ('moveLayer').

onmousemove

The first thing the script does is test to see if 'selectedLayer' has been set - assuming it has, it now resets the location parameters for the cardLayern by

adding in the change in the (x,y) co-ordinates of the mouse since the mouse last moved (or was first pressed down). Finally the recorded (x,y) co-ordinates of the mouse are updated. The browser causes this method to be triggered discretely but this happens frequently enough that the movement of the Card on the screen appears smooth to the user.

onmouseup

The onmouseup method of the document is set to a script function ('disengage') from the moment the layer is first created. The first thing the script does when called is test to see if 'selectedLayer' has been set - assuming it has it now sets selectedLayer to null and unsets the onmousemove method of the document. This gives the user the impression that the card has been 'let go'.

To improve the user's experience when moving cards on the screen the following steps are performed:

The background colour of the cardLayer changes when it is 'engaged'. The whole cardLayer can also be made for transport for moving.
The background colour changes back when is it 'disengaged'.
The z-index, which represents ranking of card images above each other, is set to a high value when the Card is engaged. This means that the Card appears above the other Cards on the screen. This may be done by tracking the highest allocated z-index value and using a z-index value one greater than the highest used to date and update max z-index variable each time this new high-level is set.
When the user drags the Card off the edge of the screen there is a risk that the onmouseup method will be missed by the script and the Card continue to move around even though the mouse has been lifted. This is countered by

tracking the edges of the browser window and forcing the 'disengage' function to be called each time the mouse crosses the edge of the window.

Re-sizing is done using the same principals as moving Cards on the screen. In this case however it is the cardRszlayern that listens for the onmousedown and the onmousemove events and the attached script function causes the cardLayern to be resized as opposed to moved. Again the same types of subtle improvements can be added (changing background colour etc.).

Dropping items on a tab or wastebin is accomplished by checking the mouse co-ordinates when the mouse button is released to see if it is within the boundaries of the wastebin or one of the Leaf Tabs. If it is over the wastebin it is deleted and if it is over a Leaf tab it is moved to the appropriate Leaf.

Updating/Modifying.

Changes may be submitted to the database incrementally (as cards are moved, dropped in the wastebin or moved to another Leaf etc.) or at the end of a session when the user is asked if they wish to save their new settings. The mechanics are the same in either case. A third approach combines those two and allows the updates to be sent incrementally but not be committed to the database until the user confirms them.

If data is sent to the server incrementally, the user does not need to wait for a response from the server before continuing, this processing goes on in the background. In either situation it is important to ensure that all the updated data has been returned to the server before the main window is closed otherwise some changes will be lost. This can be guarded against by setting the onunload method

for the BODY Element of the RUI main window to give the user the option to delay the close until the data has all been received by the server.

Two alternative processes will now be described that can be used to pass the updates back to the server (without disruptive messages on the user's screen).

1. Using a FORM GET type method on a hidden IFRAME element.

Forms use two methods of returning data to web-servers: The 'post' method, which was used earlier by the Client Interface to pass the data to be saved to the server, and the 'get' method. This latter method is used here.

When used on a form the get method passes the parameters to be returned to the server as part of the URL - it may look something like:

htttp://www.mydomain.com/cgi-bin/do-your-stuff?x=21&apples=210

This is calling the script "do-your-stuff" and passing the parameters x=21 and apples=210.

This type of URL does not have to be created by a form. If a hidden IFRAME element is created and its SRC attribute set equal to the URL of the server side script with the required parameters tagged onto the end following a '?', the server can read the parameters. Having used the cookie to confirm the identity of the user, the server side script can update their database entries accordingly.

2. Using Cookies to pass data back to the server.

Short lived cookies can pass data back to the server. These are created with an expiry time of only a few seconds which is long enough to pass the data back to the server. This is achieved by calling the server script via a hidden IFRAME. Longer lived cookies can be used to hold data being transferred back to the server thereby reducing the risk of the user session being closed abruptly before the data has all been transferred. Each domain only has a limited number of cookies available and so longer lived cookies would need very careful management.

Cards dropped in the wastebin or moving Cards to another Leaf.

When a Card is dropped in the wastebin a message is sent to the server (either immediately or at the end of the session depending on how the system is configured) telling the database to delete this Card from the User's Leaf (and hence View). If the Element, contained in the Card being deleted, is not associated with any other Card it is also deleted from the database.

When a Card is moved to another Leaf, the database is updated to change the Card's Owner Leaf. Next time that View is loaded, the Card will appear in the new Leaf. The script keeps its own record of which Leaf each card belongs to, based on when the data was first loaded and the changes the user has executed subsequently and so the data does not need to be refetched from the database when a new Leaf is displayed.

Uploading data from a user's browser based favorites/bookmark collection: In IE5 making a call, in a script, to 'window.external.ImportExportFavorites' allows the repository server to obtain a copy of the user's favorite collection. Microsoft choose to format this data

in the format of Netscape's Bookmark file. In Netscape a signed script can easily be given the permission to obtain a copy the user's bookmark file.

In either case what is received at the server is a set of bookmarks in Netscape bookmark file format. This file is an HTML file setting out the bookmarks in an HTML definition list. This is a well structured file consisting largely of <A> type links with text descriptors, that can be easily parsed and uploaded into a basic set of text based elements and cards in a repository embodying the invention.

Having described the construction and operation of preferred embodiments of the invention some points will now be described in greater detail.

The definition of meaningful collection of elements is specific to HTML and in particular HTML as it is currently defined. Different rules would be used for a different Markup Language and also new rules or modifications to the following rules may be necessary if further additions or modifications are made to the specifications of HTML. It is to be understood that the present invention is not limited to HTML or to any particular mark-up language.

The rules, whilst hard-coded in the current implementation, could be derived from the HTML DTD referred to below. This type of approach would allow application to other visual XML/SGML type applications.

In some cases, the tagName is used as a shortcut to identify the Element e.g. '<BODY>' instead of an 'Element Node with a tagName = "BODY"'. In doing so it should be noted that the tag need not always appear in the raw HTML file for the associated Element to exist within the DOM.

1. Skeletal Elements - Used to Stop Node Traversal

These are the tags that are used to stop the traversing up through the DOM Node tree. In broad terms they provide the skeleton of the document. If the script encounters either of the following of these it stops searching for a further parentNode:

<BODY>

<IFRAME>

2. Base Nodes of Meaningful Collections

The HTML4 Strict Document Type Definition defines groups of elements know as Entities identifiable as *%name*. Those that come under the following definitions form common ancestors to meaningful collections of elements. Note that one or two elements are over-ruled in the list of excluded elements below:

%fontstyle

%phrase

%special

%block

In addition the following Elements are considered meaningful:

<BODY> special case, see below

<FONT> Strictly speaking this should be ignored as a deprecated Element but it is still in very common use.

In practice, however, one or two of these may be excluded as they are not very meaningful. For example
 (within %special) is merely a forced line break or <HR> (within %block).

3. Special Cases

Some elements receive special treatment in order to capture the appropriate information. Specifically:

<MAP>, which is included within %special has no meaning without an associated <IMG>, <OBJECT> or <INPUT> - the script therefor searches for the appropriate 'partner' element.

<BODY> . The content of a BODY Element will be displayed within a DIV Element in the repository so the content is placed within a new <DIV> element instead.

Text Nodes are not elements but a parent Element is created for them that allow them to be added to the repository.

4. Non-Meaningful Elements

The following Elements are not considered meaningful and are passed over during all Node traversals, but they will be included (where possible) within the DOM subtree saved.

<DEL>, <INS> - these are used to track changes in documents.

Deprecated Elements such as <APPLET>, <CENTER>, <DIR>, <ISINDEX>, <MENU>, <S>, <STRIKE>, <U>.

Elements that only exist with the HEAD element such as <META>, <STYLE>.

<NOFRAMES>, <NOSCRIPT>. Technically these are meaningful elements but by their very nature will not be saved by the script in the latest browsers. The reason is that IE5 & NN6 support both FRAMES and SCRIPTS and so these alternate tags have no meaning in this context.

<HTML>, <HEAD>, <FRAMESET>, <FRAME> cannot be reached by the script.

Elements that exist exclusively within <TABLE>, <FORM>,<OBJECT> where not specifically allowed by other rules - this would include for example <TD>, <TBODY> or <SELECT>.

5. Excluded Elements

It is chosen to exclude <SCRIPT> elements as their content can have unforeseen effects on the behaviour of the repository.

Rules for Treatment of Special Cases

For some types of nodes the script must find associated nodes or data.

For example, if a user activates the context menu over an image map ('<MAP>') the Node returned by the context menu is actually the Node of the Map. The Map may be used by an IMG, OBJECT or INPUT elements to trigger different actions, such as moving to different parts of the page or opening specific new pages. It is therefore necessary to search these other Nodes to find the appropriate element is matched to the MAP.

For example, the collection of images in the document can be obtained from the array of image Nodes held in 'document.images' within the DOM. It is then a simple matter to scan through these to find the images using an image map and in particular the one using the image map on which the mouse was placed. OBJECT and INPUT nodes can be searched by examining the NodeList returned by a getElementsByTagName("OBJECT") or getElementsByTagName("INPUT") at the document level.

In another situation style sheets/style definitions may be needed to interpret the class attributes of nodes but the presently preferred embodiment extracts the style information of each node independently so this is not necessary. If it is chosen to capture global style settings then these can be obtained by a straightforward DOM function call.

Rules for Capturing Single or Combinations of Non-Meaningful Nodes

It was stated that '<TD>' and '<TR>' tags did not represent meaningful collections of elements. In isolation they do not, without a '<TABLE>' tag, represent well formed HTML. To the user, however, it is appealing to select rows from tables or groups of adjacent cells. It is therefore made possible to select combinations of nodes which share a common ancestor node type. For example, table data or table rows could be lifted from the table. In this situation the script would create a new ancestor of the appropriate type possibly using the formatting attributes of the actual table from which they are being selectively extracted.

For example, one or more <TD> nodes would be surrounded by a <TR> node. One or more <TR> nodes would be surrounded by a <TABLE> node or a suitable combination of <COL>, <ROW>, <TBODY> and <TABLE> nodes. To undertake the later approach will require an analysis of the elements of the TABLE and identification of which rows and columns are affected and picking out the required formatting information. If complete rows or columns are selected then row and column heading could be picked up also.

It was stated, strictly speaking, that TEXT Nodes do not represent meaningful elements. Some of the time Text Nodes will be the childNode of a text formatting Element. In this case the collection of Elements are captured at the formatting Element level. However it is quite common for text Nodes to appear independently of formatting elements, for example within a Link (or <A>) Node. The embodiment must therefore transform this type of Node into an Element in order to save and subsequently display the text. This is done by embedding the text within suitable

neutral formatting element such as a Paragraph (<P>) element.

Additionally the <BODY> element can not be saved as is within a <DIV> element. This situation is handled by extracting its childNodes and giving them a new parent Node of type <DIV>.

Facilitating, in this way, the combination, or re-characterisation, of 'independent non-meaningful' elements into one, or more, meaningful collections opens up a vast array of possibilities.

Extracting HTML from the DOM

At least 3 different techniques could be employed for extracting the pertinent data from the DOM.

The first approach described above, scans the Node subTree extracting tagName, attributes, style settings and nodeValues. The two main alternatives are to clone the Node, and its descendants, or use a non-DOM method implemented in IE (and it is believed in NN6 when it is released officially).

Cloning or Importing the subTree

The actual DOM subTree of an element can be copied, thereby eliminating the need to recreate the HTML, only to have the browser parse it back into the DOM as a copy. The structure and content of the Node and all its descendants can be copied by using a cloneNode or inportNode method of the Node in question. Using the deepClone option forces a copy of all the descendant Node data. This is not a pointer to the original subTree but, with the deepClone option set, a full copy of all its

content. This allows the Node data to transferred to the new window.

The data must then be transferred to the database on the repository server. Since there is not a means of transferring this data to the server in its native DOM form, it is necessary to 'translate' the data into its implied raw HTML in order to transfer the data as text.

If a method is developed to transmit the native DOM data to the server this approach may offer significant ease of programming and efficiency benefits over the approach described in the main body of the description.

Using the innerHTML data

Internet Explorer provides access to its own version of the implied raw HTML of a Node and its descendants in the form of the innerHTML. Because of developer pressure NN6 is likely to also include this field when it is released. This data is not within the DOM specification and should not be used if DOM compliance is considered important. Other DOM compliant browsers may not offer this field and hence their users would be barred from using this method if this data field was used.

There are efficiency benefits in using this data as it eliminates the need to extract recursively the childNode, attribute, style and nodeValue data, but it has significant drawbacks. As was described earlier 'SRC', 'HREF' and other URI type attributes must often be modified to ensure that the full path is captured in the database. If the innerHTML data field was used it would be necessary to search it for instances for 'SRC' and 'HREF' and make the suitable amendments. Ensuring that only the instances where 'SRC' and 'HREF' are used as Node

attributes would require involved logic and may well end up being less efficient than recursively extracting the information from the tree. If a suitable - robust and efficient - method was found, then it would be possible to consider the use of innerHTML in a commercial environment.

In the description of the repository user interface it was mentioned that a Hierarchical Tabular Representation with Views could be adopted. An example of such a representation is shown in Figure 11. Here, the user has previously saved five elements and has opened the repository choosing to use a simple tabular interface.

Three table headings are shown, although by configuring the site, the user can add as many as she wishes.

The individual images and their links can be re-categorised by selecting the table headings from the drop-down menus to the left of each element. Sub-categories are also available, allowing a hierarchical representation of the bookmarked elements, similar in functionality to the browsers and other online bookmark services, albeit with a visual (as opposed to text-based) representation of the bookmarked elements.

This interface to the repository can be used with the same database structure as was described earlier, but uses fewer of the customisation settings.

As has been mentioned, the invention is not limited to HTML, but is applicable to any SGML based system including visually representable XML. Many systems developers are storing 'documents' in XML format, to allow easier cross platform development, conversion from one application to

another and even embedding different types of documents within each other.

In the near future, sophisticated word processing documents and spreadsheets will become part of a web-page, and vice-versa. The distinction between web-pages written in HTML and other types of documents, now stored in XML, will become increasingly blurred.

Thus, it is therefore important to recognise that the various aspects of the invention are applicable to all types of XML as long as there is an application, such as the web browsers used or an advanced word processor, that can parse and display this information, and that there is suitable access to the DOM.

The latest versions of the main web-browsers and the specification for the DOM and CSS are anticipating the inclusion of a broader set of markup tags and data into the web-browsing context. By setting out the rules for defining meaningful elements and collections of elements, as defined by their ancestor, exclusively in terms of the DTD for the XML being parsed, the various aspects of the invention can be applied to all forms of browser parseable XML.

As long as the browser is able to parse and display the XML then it is possible to capture and store most meaningful elements.

The interface would remain the same as would most of the underlying code. However, there are some methods specified in the DOM specifically for dealing with XML that would need to be used in place of their HTML equivalents. Implementation of this would be well within the capabilities of those skilled in the art.

The Repository User Interface would be suitable to store, display and organise visually parseable XML, if provided with suitable style sheets.

Some of the special treatment of specific HTML elements, such as the resizing of elements, would not work 'out of the box' and some customisation of the application may be required for specific instances or to take advantage of some of the functionality of specific situations, such as a musical notation implementation that has sound incorporated.

Various other modifications and enhancements within the scope of the invention will occur to those skilled in the art. The invention is limited only by the scope of the claims appended hereto.

CLAIMS

1. A method of storing a portion of a mark-up language page,

comprising the steps of:

identifying, from a visual representation of the page, a portion of the visual representation of the mark-up language page to be stored;

identifying a list of candidate mark-up elements from a predefined set of elements for storage;

selecting elements from the list; and

storing the selected elements.

2. A method according to claim 1, comprising storing the selected elements in a repository accessible on-line.

3. A method according to claim 1 or 2, wherein the step of identifying a list of candidate mark-up elements comprises overlying a pointer device or cursor over the identified portion.

4. A method according to claim 3 wherein the step of identifying a list of candidate mark-up elements comprises selecting a menu and selecting from the menu a command to select the portion.

5. A method according to claim 4, wherein the menu is an Internet browser context menu.

6. A method according to any of claims 1 to 5, wherein the step of identifying a list of candidate mark-up elements comprises identifying the nodes of the Document Object

Model (DOM) which represent the identified elements, and extracting the mark up code for the identified nodes.

7. A method according to claim 6, wherein the step of identifying the nodes comprises traversing the node tree of the DOM and identifying ancestor and descendant nodes representing mark-up elements in the predefined selectable set of mark-up elements.

8. A method according to claim 7 wherein the step of traversing the node tree includes the step of establishing a list of mark-up elements from the predefined set.

9. A method according to claim 8, wherein the predefined set of elements is based on the mark-up document type definition (DTD).

10. A method according to any of claims 6 to 9, wherein the step of traversing the node tree comprises determining from a predefined rule set whether a given node represents the end of the node tree traversal in a given direction.

11. A method according to claim 10, wherein the predetermined rule set is based on the mark-up document type definition (DTD).

12. A method according to claim 6 to 11, wherein the step of identifying the nodes comprises finding related nodes.

13. A method according to any of claims 6 to 12, wherein the step of identifying the nodes comprises selecting a node representing a mark-up element excluded from the list of

candidate mark-up elements where the selected node is assigned an ancestor node representing a mark-up within the predetermined set.

14. A method according to any of claims 6 to 13, wherein the step of extracting the mark up code comprises extracting raw mark-up code.

15. A method according to any of claims 6 to 13 wherein the step of extracting the mark up code comprises extracting the document object model sub-tree from the identified nodes.

16. A method according to claim 14, wherein the step of extracting the raw mark up code comprises creating a blank mark-up code string, adding to the string the opening tag from the tag name of the node, adding any non-default attributes from the node to the code string and adding any non-default style settings to the code string.

17. A method according to any of claims 14 to 16, wherein the step of extracting the mark up code comprises passing the mark-up code represented by the node to a new document.

18. A method according to claim 17, wherein the mark-up code passed to the new document is written as a series of layers.

19. A method according to claim 17 or 18, wherein the element for storage is selected from the list of candidate elements by means of a menu or according to a rule set.

20. A method according to claim 19, wherein the step of selecting the element for storage further comprises posting a mark up code from on the new document to a repository on a remote server.

21. A method according to any preceding claim, wherein the step of storing the selected element comprises accessing a remote server, the step of accessing comprising supplying user details to the server and sending the data in the mark-up code form to be stored in the user's repository.

22. A method according to any preceding claim, wherein the each element in the list of candidate elements comprises a meaningful element or collection of elements, each element being or being associated with a mark-up tag and the list of candidate elements comprising a set of meaningful elements.

23. A method according to any preceding claim, wherein the mark-up code is HTML.

24. A method according to any of claims 1 to 22, wherein the mark-up code is XML.

25. A computer program comprising program code means for performing all the steps of any one of claims 1 to 24 when the program is run on a computer.

26. A computer program comprising program code means for performing all the steps of any one of claims 1 to 24 when the program is run within an Internet Browser on a computer.

27. A computer program product comprising program code means stored on a computer readable medium for performing the method of any one of claims 1 to 24 when the program is run on a computer.

28. A computer program product comprising program code means stored on a computer readable medium for performing the method of any one of claims 1 to 24 when the program is run within an Internet browser on a computer.

29. Apparatus for storing a portion of a mark-up language page, comprising:

means for identifying, from a visual representation of the page, a portion of the visual representation of the mark-up language page to be stored;

means for identifying a list of candidate mark-up elements from a predefined set of elements for storage;

means for selecting elements from the list; and

means for storing the selected elements.

30. Apparatus according to claim 31, wherein the storage means comprises a repository accessible on-line.

31. Apparatus according to claim 29 or 30, wherein the means for identifying a list of candidate mark-up elements comprises means movable to overlie the identified portion.

32. Apparatus according to claim 31, wherein the means for identifying a list of candidate mark-up elements comprises a selectable icon or menu item.

33. Apparatus according to claim 32, wherein the menu is an Internet browser context menu.

34. Apparatus according to any of claims 29 to 33, wherein the means for identifying a list of candidate mark-up elements comprises means for identifying the nodes of the Document Object Model (DOM) which represent the identified elements; and means for extracting the mark up code for the identified node.

35. Apparatus according to claim 34, wherein the means for identifying the node comprises means for traversing the node tree of the DOM and identifying ancestor and descendant nodes representing mark-up elements in the predefined set of mark-up elements.

36. Apparatus according to claim 35, wherein the means for traversing the node tree includes means for establishing a list of mark-up elements from the candidate list.

37. Apparatus according to claim 36, wherein the predefined set of mark-up elements is based on the mark-up document type definition (DTD).

38. Apparatus according to any of claims 34 to 36, wherein the means for traversing the node tree comprises means for determining from a predefined rule set whether a given node represents the end of the node tree traversal in a given direction.

39. Apparatus according to claim 38, wherein the predetermined rule set is based on the mark-up document type definition (DTD).

40. Apparatus according to any of claims 34 to 39, wherein the means for identifying the node comprises means for finding related nodes.

41. Apparatus according to any of claims 34 to 40, wherein the means for identifying the nodes comprises selecting means for selecting a node representing a mark-up element excluded from the predefined set of mark-up elements where the selected node is assigned an ancestor node representing a mark-up within the predetermined set.

42. Apparatus according to any of claims 34 to 41, wherein the means for extracting the mark up code comprises means for extracting raw mark up code.

43. Apparatus according to any of claims 34 to 41, wherein the means for extracting the mark up code comprises means for extracting the document object model sub-tree from the identified nodes.

44. Apparatus according to claim 42, wherein the means for extracting the raw mark up code comprises means for creating a blank mark-up code string, means for adding to the string the opening tag from the tag name of the node, means for adding any non-default attributes from the node to the code string, and means for adding any non-default style settings to the code string.

45. Apparatus according to any of claims 34 to 44, comprising means for passing the extracted mark-up code represented by the node to a new document.

46. Apparatus according to claim 45, wherein the means for passing the mark-up code comprises means for passing the mark-up code mark-up code passed to the new document as a series of layers.

47. Apparatus according to claim 45 or 46, comprising a menu or a rule set to select the element for storage.

48. Apparatus according to claim 47, wherein the means for selecting the element for storage further comprises means for posting mark up code from the new document to the storage means on a remote server.

49. Apparatus according to any of claims 29 to 48, comprising means accessing a remote server, the accessing means including means for supplying user details to the server and for sending the data in the mark-up code form to be stored in the user's repository.

50. Apparatus according to any of claims 29 to 49, wherein each element in the candidate list comprises a meaningful element or collection of elements, each element being or being associated with a mark-up tag and/or attribute and the candidate elements comprising a set of meaningful elements.

51. Apparatus according to any of claims 29 to 50, wherein the mark-up code is HTML.

52. Apparatus according to any of claims 29 to 50, wherein the mark-up code is XML.

53. An Internet browser comprising apparatus according to any of claims 29 to 52.

54. A method according to any of claims 2 to 28, wherein the identified portions are stored in a repository in a non-hierarchical form whereby a plurality of identified portions may be displayed for viewing simultaneously.

55. A method according to claim 54, wherein the repository comprises a plurality of cards, each card comprising a visual representation on screen of a stored identified portion.

56. A method according to claim 55, wherein the cards are arranged into leaves, each leaf comprising at least one card.

57. A method according to claim 56, wherein each leaf has an index tab.

58. A method according to claim 56 or 57, wherein the cards are moveable around the leaves.

59. A method according to claims 56, 57 or 58, wherein each card may form a part of one or more leaves.

60. A method according to any of claims 57 to 59, comprising arranging a plurality of leaves into views, each view comprising a set of identified mark-up language page portions and their attributes.

61. A method according to claim 60, wherein a given leaf may form a part of a plurality of views.

61. A method according to any of claims 55 to 610, wherein each card comprises a containing layer containing an outer boundary and a first sub layer containing the identified mark-up language page portion.

63. A method according to claim 62, wherein each card further comprises a second sublayer containing text fields associated with the elements to be displayed in each card.

64. A method according to any of claims 55 to 63, wherein the size of the cards is variable by a user.

65 A method according to claim 64, wherein each card further comprises a resizing layer, wherein the size of the card displayed to a user may be varied by the user.

66. A method according to any of claims 56 to 65, wherein the leaves may be customised by a user, whereby the user defines one or more leaves and the cards comprising each leaf.

67. A method according to any of claims 56 to 66 wherein each leaf comprises one or more layers.

68. A method according to any of claims 60 to 67, wherein the views may be customised by a user, whereby the user defines one or more views and the leaves comprising each view.

69. A method according to any of claims 54 to 68, comprising customising the display presented to a user by modifying style settings.

70. A method according to any of claims 54 to 69, wherein the repository is held at a remote server remote from a user and a user can view stored mark-up language page portions by accessing the remote server on-line and displaying the stored portions within a web browser.

71. A method according to claim 70, comprising a plurality of repositories, each repository bing associated with one or more users, the method comprising defining access parameters whereby access to a given user's stored mark-up language page portions may be limited to the user, available to any third party or partially restricted according to the access parameters.

72. A method according to any of claims 54 to 71, wherein the cards, leaves and stored mark-up language page portions are stored as customisable mark-up code layers.

73. A method according to claim 72, wherein the customisable mark-up code layers are HTML <DIV> or <SPAN> elements.

74. A method according to claim 72, wherein the customisable mark-up code layers are XML code layers.

75. A method according to any of claims 54 to 74, wherein the selectable mark-up language page portions are mark-up code elements corresponding to one or more of a predetermined set of meaningful elements.

76. A computer program comprising program code means for performing all the steps of any one of claims 54 to 75 when the program is run on a computer.

77. A computer program product comprising program code means stored on a computer readable medium for performing the method of any one of claims 54 to 75 when the program is run on a computer.

78. A method according to claim 1, comprising at a user terminal connected to the Internet and running an Internet Browser, wherein the mark-up language page is displayed in the browser, and the repository is at a remote server, wherein a plurality of identified portions are stored in a non-hierarchical form whereby a plurality of identified portions may be displayed for viewing simultaneously.

79. Apparatus according to claim 29, wherein the storage means comprises a repository for storing the identified portions for viewing in a non-hierarchical form whereby a plurality of identified portions may be displayed for viewing simultaneously.

80. Apparatus according to claim 79, wherein the repository comprises a plurality of cards, each card comprising a visual representation on screen of a stored identified portion.

81. Apparatus according to claim 80, wherein the cards are arranged into leaves, each leaf comprising at least one card.

82. Apparatus according to claim 81, wherein each leaf has an index tab.

83. Apparatus according to claim 81 or 82, wherein the cards are moveable around the leaves.

84. Apparatus according to claims 80, 81 or 82, wherein each card may form a part of one or more leaves.

85. Apparatus according to any of claims 80 to 84, wherein the repository comprises at least one view, each view comprising one or more leaves.

86. Apparatus according to claim 85, wherein a given leaf may form a part of a plurality of views.

87. Apparatus according to any of claims 79 to 86, wherein each card comprises a containing layer containing an outer boundary and a first sub layer containing the identified mark-up language page portion.

88. Apparatus according to claim 87, wherein each card further comprises a second sub layer containing text fields associated with the elements to be displayed in each card..

89. Apparatus according to any of claims 79 to 88, wherein the size of the cards is variable by a user.

90. Apparatus according to claim 89, wherein each card further comprises a resizing layer, wherein the size of the card displayed to a user may be varied by the user.

91. Apparatus according to any of claims 81 to 90, wherein the leaves may be customised by a user, whereby the user defines one or more leaves and the cards comprising each leaf.

92. Apparatus according to any of claims 81 to 91, wherein each leaf comprises one or more layers.

93. Apparatus according to any of claims 85 to 92, wherein the views may be customised by a user, whereby the user can define one or more views and can define the leaves comprising each view.

94. Apparatus according to any of claims 79 to 93, comprising means for customising the display presented to a user by modifying style settings.

95. Apparatus according to any of claims 79 to 94, comprising a plurality of repositories, each repository having an assigned user or group of users.

96. Apparatus according to any of claims 79 to 95, wherein the repository is held at a server remote from a user and a user can view stored mark-up language page portions by accessing the remote server on-line and displaying the stored portions within a web browser.

97. Apparatus according to claim 96, comprising means for defining access parameters whereby access to a user's stored mark-up language page portions may be limited to the user, available to any third party or partially restricted according to the access parameters.

98. Apparatus according to any of claims 79 to 97, wherein the cards, leaves and stored mark-up language page portions are stored as customisable mark-up code layers.

99. Apparatus according to claim 98, wherein the customisable mark-up code layers are HTML <DIV> elements.

100 Apparatus according to claim 99, wherein the customisable mark-up code layers are XML code layers.

101 Apparatus according to any of claims 79 to 100, wherein the selectable mark-up language page portions are mark-up code elements corresponding to one or more of a predetermined set of meaningful elements.

102 Apparatus according to claim 29, comprising, at a user terminal connectable to the Internet and running an Internet Browser, wherein the mark-up language page is displayed in the browser and the repository is at a remote server, wherein a plurality of identified portions are stored in a non-hierarchical form whereby a plurality of identified portions may be displayed for viewing in the user's browser simultaneously.

103 Apparatus according to claim 29, comprising, at a user terminal connectable to the Internet and running an Internet Browser the means for identifying and selecting the portion of the mark-up language page displayed in the browser to be stored; and at a remote server, the storage means comprising a repository for storing the identified portions, the identified portions being stored for display as sizable cards arranged in one or more leaves, the leaves being

arranged in one or more views, each view comprising one or more leaves, whereby a plurality of identified portions may be displayed for viewing in the user's browser simultaneously.

104. Apparatus according to claim 29, wherein the storage means comprises a database for storing mark up elements comprising a plurality of tables including an element data table for storing data about the mark-up elements; a card data table storing information about the display, formatting and positioning of the element data stored in the element data table; a leaf data table for storing data regarding cards which can be displayed in a common leaf; and a view data table for storing data about collections of leaves.

105. A database according to claim 104, comprising a repository data table for storing data regarding individual user repositories.

106. A database according to claim 104 or 105, comprising a groups data table for storing data about groups of users.

107. A database according to claim 104, 105 or 106, comprising a user data table for storing details of authorised system users.

108. A database according to claims 106 and 107, comprising a user group data table for storing a mapping of users to groups.

109. A database according to any of claims 104 to 109, comprising a permissions data table for storing

details of access rights of individual users or groups to data stored in other tables.

110. A database according to claim 109, wherein access rights stored in the permissions table include the extent and nature of the access users or groups have to the data to which they are granted access.

111. A database according to any of claims 104 to 110, wherein the element data table stores extracted mark-up code, and address information regarding the mark-up elements.

112. A database according to claim 111, wherein the elements data table further stores information regarding the time of creation of an entry in the elements data table and the time it was last viewed by a user.

113. A database according to any of claims 104 to 112, wherein the card data table stores the location of elements to be displayed each cards together with the display parameters.

114. A database according to claim 113, wherein the display parameters include the size of the card and its position on a display.

115. A database according to claim 112 or 113, wherein the card data table further stores text fields associated with the elements to be displayed in each card.

116. A database according to any of claims 104 to 115, wherein the leaf data table stores a leaf title for

display as a tab, and stores data regarding cards to be placed in each leaf.

117. A database according to any of claims 100 to 116, wherein the permission data table stores data associating permissions granted with one of the element table, the card table, the view table, the group table or a repository table, the user to which a permission relates, the owner of the permission and the nature of the permission.

118. A database according to claim 117, wherein the nature of the permission is selected from the group comprising an ability to read, modify, create, delete and administer data in a given table.

119. A method according to claim 1 comprising the steps of defining an element data table for storing data about the mark-up elements; defining a card data table for storing information about the display, formatting and positioning of the element data stored in the element data table; defining a leaf data table for storing data regarding cards which can be displayed in a common leaf; and defining a view data table for storing data about collections of leaves.

120. A method according to claim 119, comprising defining a repository data table for storing data regarding individual user repositories.

121. A method according to claim 119 or 120, comprising defining a groups data table for storing data about groups of users.

122. A method according to claim 119, 120 or 121, comprising defining a user data table for storing details of authorised system users.

123. A method according to claims 121 and 122, comprising defining a user group data table for storing a mapping of users to groups.

124. A method according to any of claims 119 to 122, comprising defining a permissions data table for storing details of access rights of individual users or groups to data stored in other tables.

125. A method according to claim 124, wherein access rights stored in the permissions table include the extent and nature of the access users or groups have to the data to which they are granted access.

126. A method according to any of claims 119 to 125, wherein the element data table stores extracted mark-up code, and address information regarding the mark-up elements.

127. A method according to claim 126, wherein the elements data table further stores information regarding the time of creation of an entry in the elements data table and the time it was last viewed by a user.

128. A method according to any of claims 119 to 127, wherein the card data table stores the display location of elements to be displayed each cards together with the display parameters.

129. A method according to claim 128, wherein the display parameters include the size of the card and its position on a display.

130. A method according to claim 128 or 129, wherein the card data table further stores text fields associated with the elements to be displayed in each card.

131. A method according to any of claims 119 to 130, wherein the leaf data table stores a leaf title for display as a tab, and stores data regarding cards to be placed in each leaf.

132. A method according to any of claims 119 to 131, wherein the permission data table stores data associating permissions granted with one of the element table, the card table, the view table, the group table or a repository table, the user to which a permission relates, the owner of the permission and the nature of the permission.

133. A method according to claim 132, wherein the nature of the permission is selected from the group comprising an ability to read, modify, create, delete and administer data in a given table.

~~ABSTRACT (Figure 4)~~